



Defence Research and
Development Canada

Recherche et développement
pour la défense Canada



Architecture for Autonomy

Implementation and Usage on the Raptor UGV

G.S. Broten, J.A. Collier, J.L. Giesbrecht, S.P. Monckton and D.J. Mackay
DRDC Suffield

Technical Memorandum
DRDC Suffield TM 2006-188
December 2006

Canada

Architecture for Autonomy

Implementation and Usage on the Raptor UGV

G.S. Broten, J.A. Collier, J.L. Giesbrecht, S.P. Monckton, and D.J. Mackay
Defence R&D Canada – Suffield

Defence R&D Canada – Suffield

Technical Memorandum

DRDC Suffield TM 2006-188

December 2006

Principal Author

Original signed by G.S. Broten

G.S. Broten, J.A. Collier, J.L. Giesbrecht, S.P. Monckton, D.J. Mackay

Approved by

Original signed by D. Hanna

D. Hanna

Head/Autonomous Intelligent Systems Section

Approved for release by

Original signed by Dr. Paul D'Agostino

Dr. Paul D'Agostino

Chairman/Document Review Panel

© Her Majesty the Queen in Right of Canada as represented by the Minister of National Defence, 2006

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2006

Abstract

In 2002 Defence R&D Canada changed research direction from pure tele-operated land vehicles to general autonomy for land, air, and sea craft. The unique constraints of the military environment coupled with the complexity of autonomous systems drove DRDC to carefully plan a research and development infrastructure. This infrastructure, using a Component Based Software Engineering approach, would provide state of the art tools that didn't restrict the research scope; thus allowing DRDC to pursue its long-term research goals.

DRDC's long term objectives for its autonomy program address disparate unmanned ground vehicle (UGV), unattended ground sensor (UGS), air (UAV), and subsea and surface (UUV and USV) vehicles operating together with minimal human oversight (Collectively known as UxVs). The individual systems may range in complexity from simple reconnaissance mini-UAVs to sophisticated autonomous combat UGVs. These systems, when integrated into a common command and control structure that included manned elements, can provide long endurance, low risk battlefield services.

A key enabling technology for DRDC's autonomy research is a software architecture that meets both current and future requirements. DRDC adopted the Component Based Software Engineering philosophy to develop its software architecture known as the "Architecture for Autonomy". Although a well established practice in computing science, CBSE using frameworks has only recently entered common use in the field of UxV development. For industry and government, the complexity, cost, and time to re-implement stable systems often exceeds the perceived benefits of adopting a modern software infrastructure. Thus, most persevere with legacy software, adapting and modifying software when and wherever possible or necessary – adopting strategic software frameworks only when no justifiable legacy exists. Conversely, academic programs with short one or two year projects frequently exploit strategic software frameworks but with little enduring impact. Following the 2002 focus shift DRDC found itself in a unique position where researchers could freely review past experiences and latest advances in software technology, before selecting the best path forward. The open-source movement has a significant impact on DRDC's views with respect to software development. Academic frameworks, open to public scrutiny and modification, are now available for a variety of niche specific research areas and researchers leveraged this research to maximize the benefits to its autonomy research program.

This document describes the "Architecture for Autonomy", how it meets the program's current needs and details its usage on the Raptor UGV. It also presents an argument for why this architecture should also satisfy future requirements as well.

Résumé

R & D pour la défense Canada a changé la direction de sa recherche sur les véhicules terrestres purement télé-opérés pour s'orienter vers celui de l'autonomie générale d'embarcations terrestres, aériennes et maritimes. Les contraintes spécifiques à l'environnement militaire, combinées à celles de la complexité des systèmes autonomes, ont amené RDDC à planifier prudemment son infrastructure de recherche et développement. L'infrastructure, utilisant une méthode d'ingénierie logicielle à base de composants, a produit des outils d'avant garde qui ne restreignent pas la portée de la recherche et permet ainsi de poursuivre des objectifs de recherche à long terme.

Les objectifs à long terme des programmes d'autonomie de RDDC concernent des véhicules disparates terrestres sans pilote (UGV), des capteurs au sol non surveillés (UGS), des véhicules aériens sans pilote (UAV), des véhicules sous-marins (UUV) et de surface (USV) sans équipage, opérant ensemble avec un minimum de surveillance humaine (connus collectivement sous le nom de véhicules sans pilote (UxVs). Les systèmes individuels peuvent varier en complexité allant d'un mini UAV de simple reconnaissance aux UGV de combats autonomes sophistiqués. Ces systèmes, une fois intégrés sur les champs de bataille, dans une structure de commande et de contrôle commune qui inclut les éléments avec équipage, peuvent fournir des services de longue endurance à faible risque.

Une technologie clé de mise en service pour la recherche en autonomie de RDDC est une architecture de logiciels qui répond aux besoins à la fois actuels et futurs. RDDC a adopté une philosophie d'ingénierie logicielle à base de composants pour développer son architecture de logiciels connue sous le nom « Architecture d'autonomie ». Bien qu'elle soit une pratique bien établie en informatique, l'ingénierie logicielle, qui utilise des cadres conceptuels vient récemment d'être adoptée dans le domaine de mise au point des UxV. Pour l'industrie et le gouvernement, la complexité, le coût et le temps nécessaires à implémenter des systèmes stables excèdent souvent les avantages perçus à adopter une infrastructure moderne de logiciels. C'est pourquoi la plupart persévère avec le legs des logiciels, adaptant et modifiant les logiciels quand et où cela est possible ou nécessaire, adoptant des cadres conceptuels de logiciels stratégiques seulement quand il n'existe pas de legs justifiable. À l'opposé, les programmes de formation générale, ayant des projets courts de une à deux années, exploitent fréquemment les cadres conceptuels stratégiques de logiciels mais avec peu d'impact. Après le changement de direction en 2002, RDDC s'est trouvé être dans la position particulière où les chercheurs ont pu librement étudier les expériences passées et les progrès les plus récents de la technologie logicielle, avant de sélectionner la meilleure voie à suivre. Le mouvement de source ouverte a un impact important sur les idées de RDDC concernant le développement de logiciels. Les cadres conceptuels de formation générale, ouvert à l'examen et à la modification par le public, sont maintenant disponibles pour une variété de niches spécifiques à des domaines de recherche et des chercheurs ont optimisé cette recherche pour en maximiser les avantages, au profit du programme d'autonomie.

Ce document décrit « l'Architecture d'autonomie », comment cette dernière répond aux besoins actuels du programme et détaille la manière dont elle est utilisée sur le Raptor. Ce document soumet l'argument que cette architecture devrait également répondre aux besoins futurs.

Executive summary

Architecture for Autonomy

G.S. Broten, J.A. Collier, J.L. Giesbrecht, S.P. Monckton, D.J. Mackay;
DRDC Suffield TM 2006-188; Defence R&D Canada – Suffield; December 2006.

Background: Defence R&D Canada researches Autonomous Intelligent Systems as defined by the DRDC Technology Investment Strategy (TIS) where the TIS defines autonomy as “...*automated or robotic systems that operate and interact in the complex and unstructured environments of the future battlespace*”. To achieve these goals, DRDC switched focus from tele-operated land vehicles to general autonomy for land, air, and sea unmanned vehicles (UxV). Although these UxV platforms range in complexity from the small and simple with only the most basic sensing capabilities to large, sophisticated, fully augmented platforms, all UxV platforms have similar autonomy requirements; namely, the ability to operate with minimal human supervision. This minimal human intervention requirement demands that autonomous UxVs have innate decision making capabilities, which in turn implies autonomy is implemented in symbolic, algorithmic entities using software. Autonomy, with its pervasive software foundations, requires an environment that allows researchers to pool their resources by sharing algorithms; thus allowing UxVs to be implemented in a *plug-and-play* manner. This document presents DRDC’s software architecture known as the “Architecture for Autonomy” (AFA). This architecture follows the Component Based Software Engineering philosophy (CBSE), which promotes the development of portable, modular and extensible “software” that can easily and seamlessly be transported to, and integrated into, new applications.

Results: The success of UxVs hinges upon the development of autonomous capabilities. Given the complexity of the autonomy problem, multiple researchers or research groups must pool their resources and this in turn demands a portable, modular and extensible software development approach. The “Architecture for Autonomy”, based upon the CBSE philosophy, provides the software development environment required by DRDC’s autonomy researchers. The Raptor unmanned ground vehicle (UGV), with its software developed under the AFA, required multiple researchers to concurrently and independently develop and implement autonomous capabilities. The AFA framework simplified the component development process and the component-based approach was instrumental in the successful integration of each researcher’s contribution into the overall Raptor UGV.

Significance of Results: Developed under the AFA, the Raptor UGV successfully demonstrated autonomous capabilities during a trials and highlighted the achievements of the Autonomous Lands Systems project. Subsequent research, under the Cohort ARP, has built upon the ALS foundations to give the Raptor UGV more robust and sophisticated autonomous capabilities.

Future Plans: Although only DRDC Suffield currently uses the AFA, researchers at Suffield have provided this architecture to their colleagues at Valcartier for use within their unmanned aerial vehicle program.

Sommaire

Architecture for Autonomy

G.S. Broten, J.A. Collier, J.L. Giesbrecht, S.P. Monckton, D.J. Mackay;
DRDC Suffield TM 2006-188; R & D pour la défense Canada – Suffield; décembre 2006.

Contexte : R & D pour la défense Canada effectue des recherches sur les Systèmes intelligents autonomes (SIA) où autonomie est définie par la Stratégie d'investissement en technologie de RDDC comme « *systèmes automatisés ou robotiques qui opèrent et interagissent dans les environnements complexes et non structurés des espaces de combat.* » Pour accomplir ces objectifs, RDDC a détourné son attention des véhicules terrestres télé-opérés pour focaliser sur les véhicules terrestres, aériens et maritimes sans équipage (UxV). Bien que ces plates-formes UxV varient en complexité, allant du petit et simple ayant une capacité de détection des plus basiques jusqu'aux grandes plateformes sophistiquées et de capacité intensifiée au maximum, toutes les plates-formes UxV ont des besoins similaires en autonomie dont celui d'opérer avec un minimum de surveillance humaine. Ce besoin d'intervention humaine minimale exige que les UxV autonomes aient une capacité innée de prise de décisions, impliquant ainsi que l'autonomie soit implémentée en entités symboliques et algorithmiques, au moyen de logiciels. L'autonomie ayant de puissantes fondations logicielles, exige un milieu qui permet aux chercheurs de mettre leurs ressources en commun en partageant les algorithmes, ce qui permet aux UxV d'être implémentés d'une manière *auto-configurable*. Ce document présente l'architecture logicielle de RDDC, connue comme « architecture d'autonomie ». Cette architecture respecte la philosophie d'ingénierie logicielle à base de composants (CBSE) qui promeut la mise au point de « logiciels » portables, modulaires et extensibles pouvant être transportés facilement et sans coutures dans des applications intégrées et nouvelles.

Résultats : Le succès des UxVs dépend du développement des capacités autonomes. Étant donné la complexité du problème d'autonomie, une multiplicité de chercheurs et de groupes de recherche doivent mettre leurs ressources en commun ce qui exige une méthode de développement de logiciels portables, modulaires et extensible. L'« architecture d'autonomie » basée sur la philosophie d'ingénierie logicielle à base de composants, permet le développement d'un environnement de logiciels requis par les chercheurs en autonomie de RDDC. Le véhicule terrestre sans pilote, Raptor, dont le logiciel est développé avec Architecture d'autonomie, exige qu'une multiplicité de chercheurs développe indépendamment et simultanément des capacités autonomes. Le cadre conceptuel d'Architecture d'autonomie a simplifié le processus du développement des composants et la méthode à base de composants a été instrumentale pour réussir à intégrer la contribution de chaque chercheur à l'ensemble du véhicule Raptor.

La portée des résultats : Mis au point avec l'Architecture d'autonomie, le Raptor a réussi à démontrer ses capacités autonomes durant les essais et a mis en évidence les accomplissements du projet des Systèmes terrestres autonomes. La recherche ultérieure, Cohort ARP, a bâti sur les fondations STA pour donner au Raptor des capacités autonomes plus robustes et plus sophistiquées.

Plans futurs : Bien que RDDC Suffield soit le seul à utiliser l'Architecture d'autonomie, les chercheurs de Suffield ont fourni cette architecture à leurs collègues de Valcartier pour que ces derniers l'utilisent avec le programme de véhicules aériens sans pilote.

Table of contents

Abstract	i
Résumé	ii
Executive summary	iii
Sommaire	iv
Table of contents	v
List of figures	vii
List of tables	vii
1 Introduction	1
2 DRDC's Architecture for Autonomy	3
2.1 Raptor UGV	3
2.2 Adaptation of Miro	3
2.2.1 Design Patterns	4
2.2.2 Miro Services	4
3 DRDC Components	7
3.1 Sensing	8
3.1.1 Laser Rangefinder	8
3.1.2 Stereo Vision	9
3.1.3 IMU	10
3.1.4 GPS	11
3.1.5 Wheel Odometry	12
3.2 Information Processing and Representation	12
3.2.1 Terrain Map Component	13
3.2.2 Traversability Map Component	14
3.2.3 Global Map Component	15

3.2.4	ModelServer Component	15
3.3	Planning and Goal Seeking	19
3.3.1	FindPath Component	19
3.3.2	Obstacle Avoidance Component	20
3.3.3	Pure Pursuit Component	21
3.3.4	Obstacle Detection Component	22
3.4	Decision Making	22
3.5	Vehicle Control	23
4	Utilities	25
4.1	Range Sensors	25
4.1.1	QtRange3dSensor	25
4.1.2	Sensor3dStream	26
4.1.3	Sensor3dGet	26
4.2	QtBodyViewer	27
4.3	QtEventPose	27
4.3.1	QtEventStereo	28
4.4	Maps	28
4.4.1	QtMap	29
4.4.2	LocalMap	29
4.4.3	Launcher	30
5	Effort	33
6	Conclusions	37
	Annex A: Tables of Configuration Parameters	39
	References	47

List of figures

Figure 1:	Legacy ANCÆUS based Vehicles	1
Figure 2:	The Raptor UGV	3
Figure 3:	<i>Subscribe-Publish Server</i> Design Pattern	4
Figure 4:	<i>Publish Server and Reactor</i> Design Pattern	5
Figure 5:	<i>Client</i> Design Pattern	6
Figure 6:	Key Miro and CORBA Concepts and Services	6
Figure 7:	Raptor UGV Flow Diagram	7
Figure 8:	Nodding Laser	8
Figure 9:	Digital Stereo Camera	10
Figure 10:	A Traversability Map with overlaid candidate arcs	20
Figure 11:	Illustration of the method Pure Pursuit uses to follow paths.	21
Figure 12:	A selection of candidate arcs, coloured to indicate those more desirable to follow the intended path.	22
Figure 13:	Graphical Range Display	25
Figure 14:	<i>QtBodyViewer</i> Display	27
Figure 15:	<i>QtEventPose</i> Display	28
Figure 16:	<i>QtEventStereo</i> Display	28
Figure 17:	Typical Global Terrain Map	29
Figure 18:	Typical Egocentric Local Map	30
Figure 19:	Launcher Display	31

List of tables

Table 1:	Nodding Laser Component's Published Events	9
Table 2:	Nodding Laser Component's Published Interfaces	9

Table 3:	Stereo Vision Component's Published Events	10
Table 4:	Stereo Vision Component's Interfaces	10
Table 5:	<i>Imu</i> Component's Published Events	11
Table 6:	<i>Imu</i> Component's Interfaces	11
Table 7:	<i>GPS Sokkia</i> Component's Published Events	11
Table 8:	GPS Sokkia Component's Interfaces	12
Table 9:	Wheel Odometry Component's Published Events	12
Table 10:	Wheel Odometry Component's Published Interfaces	12
Table 11:	Terrain Map Component's Published Events	13
Table 12:	Terrain Map Component's Interfaces	13
Table 13:	Polling Variables and Objects	14
Table 14:	Subscribed Events	14
Table 15:	<i>Traversability Map</i> Component's Published Events	14
Table 16:	Traversability Map Component's Interfaces	15
Table 17:	Subscribed Events	15
Table 18:	Polled Interface provided by GlobalMap for use by the FindPath component	16
Table 19:	PlatformIDL methods.	17
Table 20:	ModelServer's Subscribed Events	17
Table 21:	PoseTransformIDL Structure published by ModelServer on 'EventChannel'.	18
Table 22:	Planning and Goal Seeking Components	19
Table 23:	A vote for a single candidate arc	20
Table 24:	Polled Interface for Waypoints in <i>Pure Pursuit</i>	21
Table 25:	Decision Making Component	23

Table 26:	Vehicle Control Polling Interfaces	23
Table 27:	Vehicle Control Component	24
Table 28:	QtRange3dSensor Objects and Interfaces	26
Table 29:	Sensor3dStream Subscribed Events	26
Table 30:	Sensor3dStream Polled Objects from the Model Server	26
Table 31:	Sensor3dGet Polling Variables and Objects	27
Table 32:	Global Terrain Map Interfaces and Polled Objects	29
Table 33:	Egocentric Terrain Map Interfaces and Polled Objects	30
Table 34:	Launcher Process XML File entries. Launcher supports environment variables (e.g. \$NSC, \$NS, \$TAO_ROOT, etc.) in both Path and Argument fields.	32
Table 35:	Estimated Effort associated with Configuration and Testing	33
Table 36:	Estimated Effort to Implement Text based Utilities	34
Table 37:	Estimated Core Component Efforts	35
Table 38:	Estimated Effort to Implement Graphical based Utilities	36
Table A.1:	<i>Nodding Laser</i> Component's Configuration Parameters	39
Table A.2:	<i>Stereo Vision</i> Component's Configuration Parameters	39
Table A.3:	<i>Imu</i> Component's Configuration Parameters	40
Table A.4:	<i>GPS Sokkia</i> Component's Configuration Parameters	40
Table A.5:	<i>Wheel Odometry</i> Component's Configuration Parameters	40
Table A.6:	<i>Terrain Map</i> Component's Configuration Parameters	41
Table A.7:	<i>Traversability Map</i> Component's Configuration Parameters	42
Table A.8:	ModelServer's Configuration Parameters	43
Table A.9:	<i>FindPath</i> Component's Configuration Parameters	43
Table A.10:	<i>Obstacle Avoidance</i> Component's Configuration Parameters	44

Table A.11: <i>Obstacle Detection</i> Component's Configuration Parameters	45
Table A.12: <i>Arc Arbiter</i> Component's Configuration Parameters	45
Table A.13: <i>Vehicle Control</i> Component's Configuration Parameters	46

1 Introduction

DRDC, a defence research organization with a 20 year history of application development in tele-operated air and land vehicles, has developed numerous tele-operated unmanned ground vehicles (UGV), many founded on the ANCÆUS command and control system. The ANCÆUS control network used Motorola 68HC16 microcontrollers and an IBM compatible computer, running OS/2, implemented the control station. A selection of ANCÆUS based vehicles are shown in Figure 1.



Figure 1: *Legacy ANCÆUS based Vehicles*

With DRDC's 2002 switch in focus to autonomy a review of ANCÆUS 's status was required [1]. This review concluded that ANCÆUS is fundamentally a command interface and was inadequate for autonomous applications. Thus DRDC researchers, using previous experiences with tele-operated vehicles and software, thoroughly reviewed the current state-of-the-art in autonomy. This review concluded that future autonomous systems require a portable, modular and extensible architecture "... that, at once, supports and encourages distributed computing, *and* frees investigators to focus on the development of intelligent single and multi-vehicle control systems." [2] It also foresaw an architecture that, ideally " ... should seamlessly transition between real vehicle control, system diagnosis through the replay of gathered data and the control of a vehicle in a simulated world." This approach would free the investigator to develop intelligence algorithms without the distractions caused

by real vehicle implementations.

To facilitate the dissemination of research this architecture would follow an open source philosophy and use open tools, applications and operating systems [3]. These open source tools and applications include

- the Linux operating system,
- the GNU toolchain including make, autoconf and the gcc compiler and
- open source libraries: GSL, ATLAS, Lapack, Boost and others.

Combatting a “must be invented here” tradition, DRDC readily examined the current trends in robotics and quickly realized a framework approach could mitigate many risks and relieve many constraints incurred by pure in-house software development. This review of architectures and frameworks quickly produced a short list of architectures that were capable of meeting DRDC’s autonomy requirements. Of the five candidate architectures, Player/Stage [4], Carmen [5], Marie [6], Miro [7, 8] and Orca [9], Miro contained the feature set that best matched DRDC needs [10, 11]. Thus, it was concluded that the ACE/TAO/MIRO toolchain [12, 13], with powerful CORBA [14, 15] interprocess communication, provided the most open, proven, real-time, and portable code base for a military research program.

This report is organized into 6 sections. Section 2 gives an overview of the “Architecture for Autonomy” and its implementation. In Section 3 the components developed under the AFA are detailed. Utilities that assist in the visualization and monitoring of system/component status are described in Section 4. The effort required to develop autonomous capabilities is summarized in Section 5. Finally, Section 6 provides conclusions.

2 DRDC's Architecture for Autonomy

Scale, modularity, distribution, platform independence, and flexibility govern military applications of autonomous unmanned vehicle control. The software environment must scale to tolerate different team sizes and must be modular to support variation in payloads. It must accommodate distributed computing capabilities through networking and must separate hardware from software to achieve platform independence. Finally, the software must not limit engineers to any single method of vehicle control. To meet the above requirements DRDC adapted the open source Miro [16] framework.

2.1 Raptor UGV

DRDC adapted and extended Miro to support the Koyker Industries Raptor, an Ackerman steered, hydrostatic, all wheel drive utility vehicle. Shown in Figure 2 is the Raptor vehicle XJ Designs modified such that throttle, brakes, steering, and other parameters could be monitored and controlled through a single software interface. Also mounted on this vehicle are sensors for outdoor, autonomous operations.



Figure 2: *The Raptor UGV*

The sensing systems include:

- Proprioceptive sensing collecting raw position and orientation data via a Gps, Imu and odometry.
- Exteroceptive sensing including laser rangefinders [17, 18] and stereo vision.

2.2 Adaptation of Miro

Although the Miro framework was originally developed for a robosoccer application, it was easily adapted to unmanned ground vehicle applications. Miro's customized system infrastructure layer, based upon the TAO [19] real-time CORBA [14] implementation using the

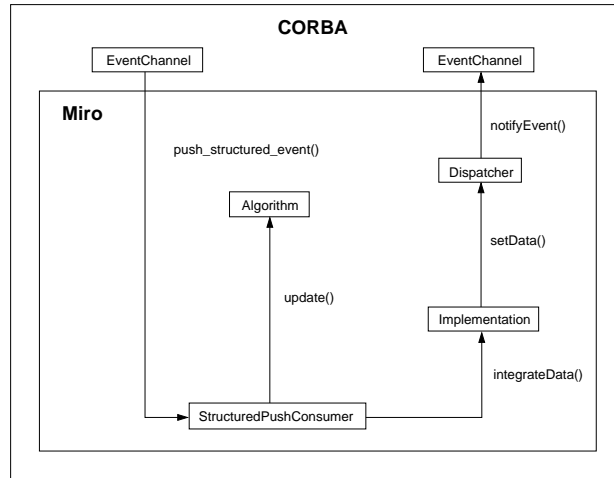


Figure 3: *Subscribe-Publish Server Design Pattern*

ACE toolkit [20], provides blackbox services such as mechanisms for logical communications, concurrency and device abstraction. These Miro services facilitate the application development by following standard *design patterns*.

2.2.1 Design Patterns

Miro uses three basic design patterns [21] to implement services, as shown in Figures 3 through 5. These design patterns include:

Subscribe-Publish Server (SPS), shown in Figure 3, receives events, processes the data and publishes events. Using this design pattern a Miro server becomes an independent component with its interfaces defined by the CORBA objects that facilitate event subscription and publication, as well as the CORBA interfaces that enable polling.

Publish Server and Reactor (PSR), shown in Figure 4, serves as the basis for handling external hardware/software entities. Its fundamental objective is to separate the physical device driver from the software that publishes events and the interfaces that enable polling.

Client, shown in Figure 5, is the design pattern that allows a client process to poll data from a Miro server.

2.2.2 Miro Services

Implementing an application, using the Miro design patterns described in Section 2.2.1, requires the understanding of several key CORBA concepts and Miro services. Figure 6 shows the relationships between the CORBA concepts, the Miro Server and Client services.

The three CORBA concepts and two Miro services are:

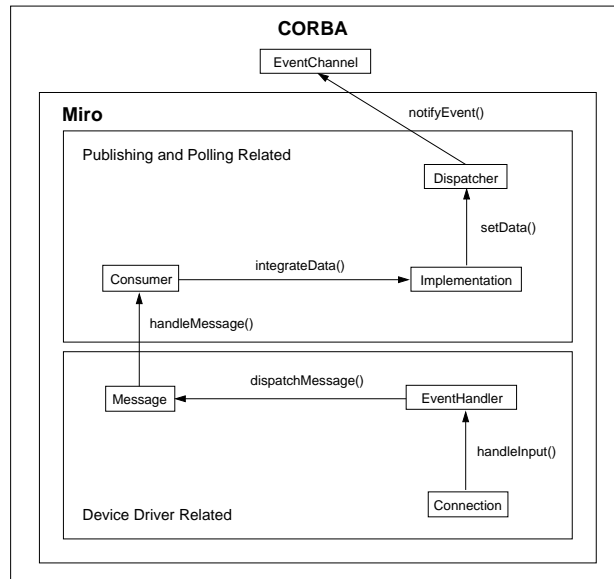


Figure 4: *Publish Server and Reactor Design Pattern*

CORBA Naming Service: Much like a phone book's white pages that maps names to phone numbers, the CORBA Naming Service facilitates data exchange between processes by mapping object names and event channel names.

CORBA Event Channels and Polling: Using CORBA capabilities Miro implements both the message and information paradigms, as polling and event channels respectively. Under the polling paradigm a Miro client, using CORBA interfaces, directly requests and receives data from a Miro server. Event channels, implemented using the CORBA Notification Service, allow a Miro server to anonymously subscribe to events published by another Miro server.

Interface Description Language: The Interface Description Language (IDL) describes the properties of a CORBA object in terms of data types and access methods known as interfaces. Thus, via the IDL language, a CORBA object represents a component with an interface that allows independent processes to exchange information in a network transparent manner.

Miro Server: Using the Miro server framework, both traditional client-server transactions and event driven processing can occur simultaneously. The Miro server is implemented under the Subscribe-Publish Server design pattern or the Publish Server and Reactor design pattern.

Miro Client: The Miro client defines a framework, using the client side IDL object binding implemented under the Client design pattern, that allows a client process to poll a Miro server following the familiar CORBA client-server transaction model.

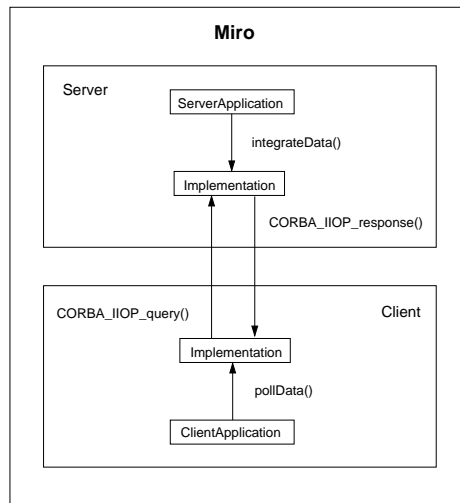


Figure 5: *Client Design Pattern*

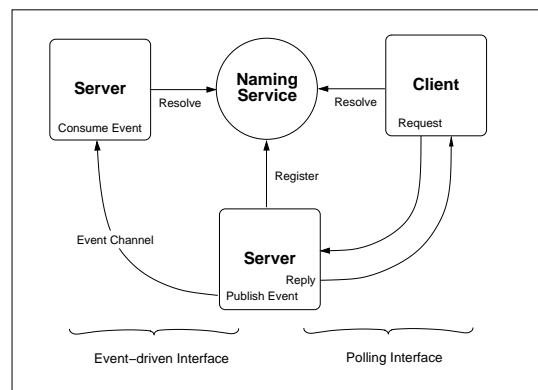


Figure 6: *Key Miro and CORBA Concepts and Services*

3 DRDC Components

Using a white-box approach researchers created UGV specific components using the Miro framework. These components were implemented under one of the following Miro design patterns: *Subscribe-Publish Server*, *Publish Server and Reactor* and *Client*. The modular vehicle controller exploits ACE/TAO/CORBA capabilities to run several behaviour components such as *Goal Seeking*, *Obstacle Detection*, *Obstacle Avoidance*, and *Path Planning*, independently, on one or more CPUs, distributed across a TCP/IP network to achieve varying levels of autonomy. Figure 7 shows a flow diagram of the architecture implemented on the Raptor UGV. Each solid box represents a Miro based component that can reside anywhere on a network and each connector represents data flows accessible to any subscriber service.

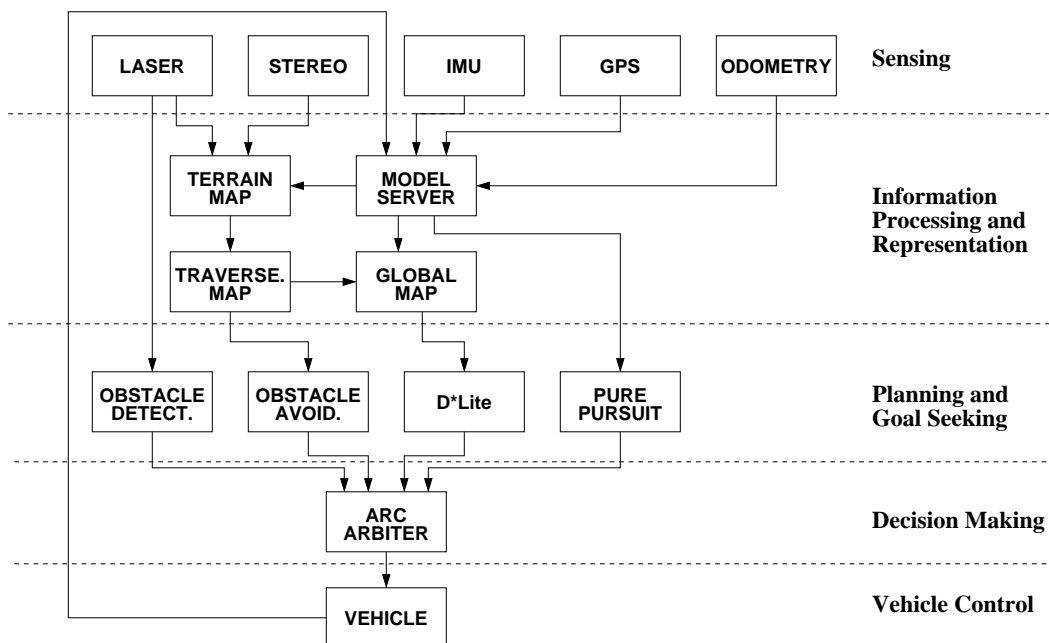


Figure 7: Raptor UGV Flow Diagram

The Raptor UGV components are divided into logical domains:

- Sensing
- Information Processing and Representation
- Planning and Goal Seeking
- Decision Making
- Vehicle Control
- Utilities

The following sections, covering each logical domain, provide details on each component's implementation.

3.1 Sensing

Under the ALS demonstration configuration the Raptor UGV received data from five unique sensing devices. Four of these devices, laser ranging, stereo vision ranging, IMU, and GPS, are dedicated sensors; wheel odometry is provided by the Raptor vehicle.

The AFA philosophy requires that each sensor interface use a component based approach, where raw data is acquired, converted to a generic data representation, packaged into an IDL defined structure, and published anonymously as events to be consumed by interested parties. Additionally, all sensing components respond to asynchronous poll requests for data. The following sections detail each sensing component's implementation, with specific references to their underlying design pattern and to their interfaces.

3.1.1 Laser Rangefinder

DRDC, in conjunction with Scientific Instrumentation Ltd., developed a nodding device for the SICK LMS 211 laser. The SICK laser measures the time of flight for a laser light pulse and internally converts the time value into the corresponding distance. To ensure timeliness, RTEMS, a real-time operating system is used as the operating system for the on-board controller. Figure 8 shows the custom nodding mechanism with a SICK laser rangefinder.



Figure 8: *Nodding Laser*

The *Nodding Laser* component, based upon the PSR design pattern, uses an ethernet connection to retrieve raw range data from the SICK laser, processes the data into a 3-D format and publishes this data using the Range3dLaserIDL/Range3dSeqEventIDL structures. The interface with the physical world builds upon the ACE reactor, as it simplifies TCP/IP based interprocess communications. The Miro server aspect of this component allow it to concurrently publish events while responding to poll requests.

Table 1 lists the events that could be published by the *Nodding Laser* component and their respective update rates. As can be seen in this table range events may be, either fixed size, static arrays or variable length CORBA sequences.

Published Events	Implementation	Update Rate
Range3dLaserIDL	Static Array	26.6 ms
Range3dSeqEventIDL	CORBA Sequence	26.6 ms

Table 1: *Nodding Laser Component's Published Events*

The *Nodding Laser* component also presents CORBA compliant interfaces that support network transparent polling. A list of these polling interfaces is given in Table 2. As with the publication of events, the polling interfaces supports both fixed size arrays and dynamic sequences.

Poll Interface	Polled Event	Implementation
get3dLaserFullScan()	Range3dLaserIDL	Static Array
get3dLaserWaitFullScan()	Range3dLaserIDL	Static Array
get3dSeqFullScan()	Range3dSeqEventIDL	CORBA Sequence
get3dSeqWaitFullScan()	Range3dSeqEventIDL	CORBA Sequence

Table 2: *Nodding Laser Component's Published Interfaces*

This component registers event and interface names with the CORBA Naming Service (see Section 2.2.2) using the defaults provided by the module's configuration file. Given the Raptor vehicle supports multiple forward looking nodding lasers, and backwards directed nodding lasers, the root event/interface name commonly starts with *Laser* and is concatenated with a unique I.D. number¹. This parameter, along with others, are read at run-time and defined by the *NodLaserConfig* section in the Raptor's XML configuration file. Each parameter, along with its significance, is listed in Table A.1 of the appendix.

3.1.2 Stereo Vision

The Raptor UGV uses the commercial Digiclops camera, produced by Point Grey Systems, as a stereo vision device. The Digiclops features a trinocular camera arrangement and is capable of producing disparity maps with a resolution of 640×480 , but on the Raptor UGV the Digiclops' performance is downgraded to a resolution of 320×240 . Figure 9 shows the Digiclops stereo system. The *Stereo Vision* component is derived from the PSR design pattern; thus, it features both an ACE reactor and a Miro server. The ACE reactor implements *schedule_timer()* and *handle_timeout()* functions that allow images to be acquired, the disparity processed and the event publication process to occur at a user defined rate. The component's Miro server allows the event publication process to running concurrently with poll requests.

¹Such as *Laser_11* or *Laser_12*

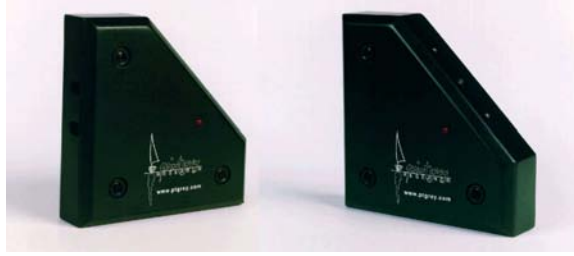


Figure 9: Digital Stereo Camera

Table 3 lists the events that could be published by the *Stereo Vision* component and their respective update rates. As can be seen in this table range events may be, either fixed size, static arrays or variable length CORBA sequences. The fixed size arrays must be specified at compile time, whereas CORBA sequences are flexible and can be specified at run-time.

Published Events	Implementation	Update Rate
Range3dStereoEventIDL	Static Array	Variable, default 2 sec.
Range3dArrayEventIDL	CORBA Sequence	Variable, default 2 sec.

Table 3: Stereo Vision Component's Published Events

The *Stereo Vision* component also presents CORBA compliant interfaces that support network transparent polling. A list of these polling interfaces is given in Table 4. As with the publication of events, the polling interfaces for this component supports both fixed size arrays and dynamic sequences.

Poll Interface	Polled Event	Implementation
get3dStereoFullScan()	Range3dStereoEventIDL	Static Array
get3dStereoWaitFullScan()	Range3dStereoEventIDL	Static Array
Range3dArrayEventIDL()	Range3dArrayEventIDL	CORBA Sequence
get3dArrayWaitFullScan()	Range3dArrayEventIDL	CORBA Sequence

Table 4: Stereo Vision Component's Interfaces

Commonly, the *Stereo Vision* component publishes data under the *Stereo* event name. The *Stereo* name is also used to register its interfaces with the CORBA Naming Service. This name, along with other configuration values, is defined in the Raptor's XML configuration file, under the *StereoConfig* section. A complete list of configuration parameters is given in Table A.2, which is located in the appendix.

3.1.3 IMU

The Raptor UGV uses the Microstrain, 3DM-GX1 IMU, gyro enhanced orientation sensor to provide orientation data. The *IMU* component, derived from the *Publish Server* and

Reactor design pattern, acquires orientation data from the device using serial communications and publishes it as a ImuIDL event, as shown in Table 5. Encoded within this event is the vehicle's orientation and the accelerations experienced.

Published Events	Implementation	Update Rate
ImuIDL	Structure of Doubles	100msec

Table 5: *Imu Component's Published Events*

Upon startup the component's event and interface names are registered with the CORBA Naming Service. The event and interface names default to *Imu*, though this name can be changed via the configuration file. The interface registration enables polling requests, thus allowing external components to access the most recent orientation via the callback function shown in Table 6.

Poll Interface	Polled Event	Implementation
getImuData()	ImuIDL	Structure of Doubles

Table 6: *Imu Component's Interfaces*

The *Imu*'s run-time operation is controlled by the parameters defined in the Raptor's XML configuration file. Each parameter, along with its significance is listed in Table A.3, located in the appendix.

3.1.4 GPS

A Sokkia GSR2600 receiver and a Pacific Crest PDL radio provide differentially corrected GPS localization for the Raptor vehicle at an update rate of 4 Hz, via a serial port interface. The *GPS Sokkia* component provides communications with the GPS hardware, and publishes GpsIDL events; making this information available to other software components. The *GPS Sokkia* component is derived from the PSR design pattern, using the ACE Reactor for serial port control, while the Miro Server publishes events and responds to poll requests. The GpsIDL event, shown in Table 7, contains a variety of data items, such as latitude, longitude, elevation, eastings, northings, satellite count, horizontal speed, etc.

Published Events	Implementation	Update Rate
GpsIDL	Structure of Doubles,Integers	250msec

Table 7: *GPS Sokkia Component's Published Events*

On startup the *GPS Sokkia* component will register a *Gps* interface with the CORBA Naming Service, but, if necessary this name can be changed via the configuration file. This registration process allows external components to resolve the component's interfaces, as listed in Table 8, and poll for position data.

Poll Interface	Polled Event	Implementation
getPosition()	GpsIDL	Structure of Doubles,Integers

Table 8: GPS Sokkia Component's Interfaces

Finally, *GPS Sokkia* component's run-time operation can be configured by the parameters defined in the Raptor's XML configuration file. The parameters, located under the *GpsSokkiaConfig* section, are given in Table A.4 of the appendix. This table lists each parameter and provides a description of its significance and usage.

3.1.5 Wheel Odometry

The Raptor UGV obtains odometry information from its on-board MPC555 controller, which measures and accumulates data from the two front wheel encoders. This odometry data is transmitted via a serial interface for consumption by the *Wheel Odometry* component. This component is derived from the PSR design pattern, and thus, it incorporates both an ACE reactor and a Miro server. The component processes serial data using the ACE reactor callback function, concurrently publish WheelDataIDL events and respond to poll requests. Table 9 specifies the event published by the *Wheel Odometry* component. The component's polling interface is detailed in Table 10.

Published Events	Update Rate
WheelDataIDL	250 ms (configurable)

Table 9: Wheel Odometry Component's Published Events

Poll Interface	Polled Event	Implementation
getWheelData()	WheelDataIDL	Static Array

Table 10: Wheel Odometry Component's Published Interfaces

The event and interface names registered with the CORBA Naming Service are defined in the module's configuration file. Commonly, the *Wheel Odometry* component publishes WheelDataIDL data under the *RaptorOdometry* event name and uses the same name for resolving interfaces.

The runtime parameters, which affect the wheel odometry, are defined in the Raptor XML file, and are listed in Table A.5 of the appendix.

3.2 Information Processing and Representation

Before the Raptor UGV can make decisions it must sense the environment and use this information to create applicable representations. The *Terrain Map*, *Traversability Map*, and *ModelServer* components all acquire data, process this information and produce new representations that are useful for autonomous operations.

3.2.1 Terrain Map Component

The *Terrain Map* component receives data from the exteroceptive sensors, such as the SICK laser and Digiclops stereo camera and fuses it into a grid map. The grid map, formed by rectangular array of regions, serves as the primary level world representation². The *Terrain Map* component, derived from the *Subscribe-Publish Server* design pattern, can concurrently publish events, receive events and respond to poll requests. Table 11 lists the events that could be published by the *Terrain Map* component and their respective update rates. As can be seen in this table, terrain map events may be fixed size, static arrays or variable length CORBA sequences.

Published Events	Implementation	Update Rate
MapArrayEventIDL	Static Array	Variable, default 1 sec.
MapSeqEventIDL	CORBA Sequence	Variable, default 1 sec.
EgoMapArrayEventIDL	Static Array	Variable, default 1 sec.
EgoMapSeqEventIDL	CORBA Sequence	Variable, default 1 sec.

Table 11: *Terrain Map Component's Published Events*

The *Terrain Map* component also presents CORBA compliant interfaces that support network transparent polling. A list of these polling interfaces is given in Table 12. As with the publication of events, the polling interfaces support both fixed size arrays and dynamic sequences.

Poll Interface	Polled Object	Implementation
getMapArray()	MapArrayEventIDL	Static Array
getWaitMapArray()	MapArrayEventIDL	Static Array
getMapSeq()	MapSeqEventIDL	CORBA Sequence
getWaitMapSeq()	MapSeqEventIDL	CORBA Sequence
getEgoMapArray()	MapArrayEventIDL	Static Array
getWaitEgoMapArray()	MapArrayEventIDL	Static Array
getEgoMapSeq()	MapSeqEventIDL	CORBA Sequence
getWaitEgoMapSeq()	MapSeqEventIDL	CORBA Sequence

Table 12: *Terrain Map Component's Interfaces*

Event and interface names are registered with the CORBA Naming Service using the values defined in the module's configuration file. Commonly, the *Terrain Map* component publishes MapArrayEventIDL/MapSeqEventIDL events under the *Terrain* event name. The *Terrain* name is also used to resolve the component's interface callback functions.

Upon start-up the *Terrain Map* component also polls the *ModelServer* component for geometry information regarding the placement of the range devices. Using this geometry

²See Figure 17, in Section 4.4 for a typical terrain map.

information the map transforms the range data to the map's frame of reference. Table 13 shows the variables required to poll the *ModelServer* component and the transformation object returned. Additionally, the *Terrain Map* component requires range data and pose data to create the map; thus it typically subscribes to the events listed in Table 14.

Object Server	Resolution Variable	Polled Interface	Polled Object
ModelServer	Platform_var	getTransformation()	PoseTransformIDL_var

Table 13: Polling Variables and Objects

Event Channel	Event Name	Event Object
EventChannel	Laser11	Range3dLaserIDL/Range3dSeqEventIDL
EventChannel	Laser12	Range3dLaserIDL/Range3dSeqEventIDL
EventChannel	Stereo	Range3dStereoEventIDL/Range3dArrayEventIDL

Table 14: Subscribed Events

The *MapConfig* section of the Raptor's xml configuration file specifies the *Terrain Map* component's run-time operation. These parameters specify numerous options, ranging from subscribed event names, published event names, map depth and width, to the definitions of various update periods. Table A.6 provides a complete list of these parameters along with their significance and default values.

3.2.2 Traversability Map Component

The *Traversability Map* component subscribes to the events listed in Table 17, which are published by the *Terrain Map* component described in Section 3.2.1. This component converts the fine resolution terrain map to a coarser traversability map. The traversability map is also grid based where each element contains measures of traversability, the ease with which a UGV can navigate the given cell, and *Goodness*, the accuracy of the data used to produce traversability. The *Traversability Map* component is derived from the SPS design pattern and consequently, it can concurrently publish events and respond to poll requests. Table 15 lists the possible events that could be published by the this component and their respective update rates.

Published Events	Implementation	Update Rate
TravMapArrayEventIDL	Static Array	Variable, default 1 sec.
EgoMapArrayEventIDL	Static Array	Variable, default 1 sec.

Table 15: Traversability Map Component's Published Events

The *Traversability Map* component, derived from the Miro server, also provides the polling interfaces listed in Table 16. As with the publication of events, the polling interfaces are implemented using fixed size arrays. *Traverse* is the default name for registering events and interfaces with the CORBA Naming Service. These names, along with other parameters,

are specified under the *TraverseMap* section of the Raptor’s xml file. Table A.7, located in the appendix, specifies all configurable run-time parameters and describes each parameter’s significance.

Poll Interface	Polled Object	Implementation
getTravMapArray()	TravMapArrayEventIDL	Static Array
getWaitTravMapArray()	TravMapArrayEventIDL	Static Array
getEtravMapArray()	EtravMapArrayEventIDL	Static Array
getWaitEtravMapArray()	EtravMapArrayEventIDL	Static Array

Table 16: *Traversability Map Component’s Interfaces*

Event Channel	Event Name	Event Object
EventChannel	Terrain	MapArrayEventIDL
EventChannel	Terrain	EgoMapArrayEventIDL

Table 17: *Subscribed Events*

3.2.3 Global Map Component

The *Global Map* component maintains a cumulative view of the traversability information generated by the *Traversability Map* component, described in the previous section. The *Global Map* component is derived from the *Subscribe-Publish Server* design pattern. It subscribes to TravMapArrayEventIDL events published by the *Traversability Map* component and generates PlanMapEventIDL events. The PlanMapEventIDL events simply encapsulate the information contained in the TravMapArrayEventIDL events received by *Global Map* component and provide a convenient means to trigger the redisplay of the developing global map in the *qtEventPlanMap* viewer.

In addition, the *Global Map* component employs a polled interface for use by the *FindPath* component in planning a path. This interface enables

- waypoints for a path to be set and queried in the same fashion as the *PurePursuit* component handles waypoints, described in Section 3.3.3,
- a planning space boundary to be set and queried, and
- the vehicle’s current location and the cost of traversal to adjacent locations in the planning space to be queried.

Table 18 details the polling interface provided by *Global Map* component.

3.2.4 ModelServer Component

Large vehicle navigation services rely on widely distributed components, particularly sensors, positioned both in local and global coordinates. To establish these positions, re-

Poll Interface	Description
boolean valid(in PlanPointIDL p)	Returns true if the planning node addressed by p is within the planning grid and reachable; false otherwise.
double getPlanningMap(in PlanPointIDL p) double getSensingMap(in PlanPointIDL p)	Accessors returning the cost, <i>a priori</i> and sensed, respectively, of moving to p from an adjacent point in the planning grid.
boolean setPlanningMap(in PlanPointIDL p) boolean setSensingMap(in PlanPointIDL p)	Accessors setting the cost of moving to point p from an adjacent point in the planning space; returns true if successful, false otherwise.
PlanPointIDL getMap(in PlanPointIDL p) boolean setMap(in PlanPointIDL p)	Accessors returning/setting the PlanPointIDL structure associated with the point p in the planning space; setMap(.) returns true if successful, false otherwise.
PoseTransformIDL getTransformToPlanningFromWorld() PoseTransformIDL getTransformToWorldFromPlanning()	Accessors returning the transforms between the world and planning frames.
boolean getVehiclePosition(out Position3DIDL p)	Accessor returning the vehicle's current position as reported by the latest TravMapArrayEventIDL received by <i>Global Map</i> ; returns true if successful, false otherwise.
boolean getNextWaypoint(out WaypointIDL point, in WaypointIDL current)	Accessor returning the next waypoint; returns true if successful, false otherwise.

Table 18: Polled Interface provided by *GlobalMap* for use by the *FindPath* component

searchers combined an internal geometry database and vehicle pose estimation into a *ModelServer* that publishes both fixed local geometric transforms and global vehicle position in UTM coordinates.

Internal Geometry Database: The internal geometry database follows common dynamic modeling conventions (e.g. in ODE[22]) by managing three data types: a *Body*, a *BodyFrame*, and a *Constraint*, accessible through a *BodyList*, a *ConstraintList*, and a directed graph *Model*. *Constraints* bind distinct *Body-BodyFrame* pairs through a time invariant homogeneous transform and pointers to *From* and *To* bodyframes respectively. The *Model* then resembles a cyclic directed graph of *Body-BodyFrame* vertices with transformation connectors

PlatformIDL methods, summarized in Table 19, provides the primary poll interface to *ModelServer* geometry. Through this interface, clients can interrogate the model for:

- all available *Bodies* returned as a sequence of strings.

Returned Type	Function Call	Returns
PoseTransformIDL	getTransformation (A,B)	transform ${}^A\mathbf{T}_B$.
StringSequenceIDL	getBodyList()	all body names in model.
StringSequenceIDL	getBodyFrameList()	all frame names in body.

Table 19: PlatformIDL methods.

Event Channel	Event Name	Event Object
EventChannel	Gps	GpsIDL
EventChannel	Imu	ImuIDL
EventChannel	Odometry	WheelEventIDL

Table 20: ModelServer’s Subscribed Events

- all available *Frames* for any body returned as a sequence of strings.
- transforms between any two *Body/Bodyframe* nodes.

To poll ModelServer, a client design pattern follows a structure similar to Figure 5.

Geometry queries from any given process need occur only once since internal vehicle geometry does not change over time.

Pose Estimation: Unlike the simple client server model of the geometry database, the pose estimation engine of ModelServer exemplifies a *Subscribe-Publish server* design pattern, consuming and fusing *GPS*, *IMU* and *odometry* into published *pose* events sent to numerous subscribers³. In ModelServer, an object derived from Miro’s Structured PushConsumer class responds to every published GPS, IMU, and odometry event (e.g. published by SokkiaService, the GPS server), described in Table 20, via the `push_structured_event()` method. In turn, this method calls a localization filter, effectively updating the location estimate with each new sensor event.

ModelServer publishes this new location estimate as a *Pose* event corresponding to the *PoseTransformIDL* data structure. Clients subscribing to ModelServer *Pose* events use a similar design pattern to ModelServer itself.

The PoseIDL in Table 21 provides type definitions for the PlatformIDL object in Table 19, the most significant being *PoseTransformIDL*. This time stamped structure contains a pose validity flag to warn consumers of unreliable data, an initial UTM coordinate, and a homogeneous transform. The validity flag provides consumers with an indication of the event’s reliability. When the vehicle is stationary, GPS heading becomes erratic and contaminates the filtered solution; thus, this flag warns clients that the pose might be questionable.

³As shown in Figure 7, the *Terrain Map*, *Global Map* and *Pure Pursuit* components subscribe to pose events.

Type	Name	Description
TimeIDL	time	Time Stamp
double[4][4]	HTransform	a 4x4 homogeneous transform.
double[3]	initialUTM	initial UTM position.
char	poseValidFlag	the Pose Valid flag.

Table 21: *PoseTransformIDL Structure published by ModelServer on 'EventChannel'.*

Since world coordinates are in UTM, vehicle position was expressed in very large meter displacements from a UTM zone origin and these large numbers often causing downstream numerical errors in the *Terrain Map* service. To prevent these errors, yet retain the true world coordinate, the *Pose* event stores a startup UTM position in `initialUTM`. `HTransform`'s position vector is measured from this point and updated as the vehicle moves.

Configuration: Modelserver's behaviour may be modified by changing variables within a configuration file briefly documented in Table A.8 The configuration file permits users to specify the name and IDL-type of subscribed, published, resolved, and offered names (those names resolvable by clients). This permits maximum flexibility in establishing communications between processes at runtime. The file also provides a mechanism to use unfiltered or 'raw' position solutions – a good sanity check during filter tuning.

Vehicle Geometry is defined within an *assembly* XML file and a number of *body* XML files (see [?] for more details). The 'ModelFilename' identifies the root *assembly* file.

Since many downstream processes do not need the high data rates provided by a pure event driven publication model, Modelserver provides a timer to send *Pose* events at a nearly constant rate regardless of input rates. The rate is set by *UpdatePeriod*, the period in milliseconds between updates.

Without magnetometers, IMU drift corrupts orientation estimates. For the Microstrain IMU, used in the initial ALS project, the drift or bias rate could be captured to correct drift from subsequent sensor readings. Since 'bias capture' could only be performed on a stationary sensor, ModelServer watched for stationary GPS positions and, after a short 'fidget delay', captured the bias rate every 'fidget period', and ceased capture if the vehicle moved. Both fidget delay and period can be set in this file.

Runtime: At runtime, ModelServer loads the model XML file and accepts client queries through a CORBA IDL interface to retrieve a Body-BodyFrame pair or a relative transformation between a *frame of reference* and a target *frame of interest*. Simultaneously, ModelServer subscribes to GPS, IMU, and WheelEvent processes, publishing a *Pose* event for every received event or at a fixed update rate.

3.3 Planning and Goal Seeking

Planning and goal seeking behaviours are implemented by the *FindPath*, *Obstacle Avoidance* and *Pure Pursuit* components. *FindPath* implements the D*Lite [23] path planning algorithm, an incremental heuristic search method implementing goal-directed robot navigation in unknown terrain. Similar to other candidate arc systems [24, 25], *Obstacle Avoidance* estimates the cost of driving candidate angles based on the traversability map. The Pure Pursuit algorithm [26], a proven method of path following, attempts to follow straight line segments between waypoints. Finally, the *Obstacle Detection* component works as a safety catch for the other modules, checking for any dangerous obstacles directly in the vehicle's immediate path.

Component	Design Pattern	Subscribed Events	Published Events	Update Rate
FindPath	PSR	-	ArcVote	≥ 500 ms
Obstacle Avoidance	SPS	TravMapArray	ArcVote	500 ms
Pure Pursuit	SPS	Pose	ArcVote	100 ms
Obstacle Detection	SPS	Range3dLaserIDL	ArcVote	26.6ms

Table 22: Planning and Goal Seeking Components

Each of these Planning and Goal Seeking components is derived from the Miro server framework. The *FindPath* component uses the PSR design pattern while the *Pure Pursuit* and *Obstacle Avoidance* components use the SPS design pattern. These components subscribe to various sensor and processing components, as shown in Figure 7 and listed in Table 22. However, each of them publishes the same type of event, an *ArcVote* event based upon the ArcVoteIDL, on the event channel.

Each *ArcVote* event expresses the behaviour's desire to travel on each arc from a pre-defined set of candidate arcs (i.e. vehicle steering angle and speeds). Each behaviour can also veto any arc that it sees fit, to ensure that the vehicle will not travel that path. It also provides a maximum speed that the vehicle should be allowed to travel. The votes from the various active behaviours will be combined in the decision making process, described in the next section, into a vehicle action. The structure of the *ArcVote* interface for a single arc is shown in Table 23. The *ArcVote* contains an array of these votes, one for each of the candidate arcs, as well as an indication of which behaviour generated this *ArcVote* event.

3.3.1 FindPath Component

The *FindPath* component doesn't subscribe to any events, instead it obtains all of the information necessary to plan a path via polling the *Global Map* component. Having planned an initial path starting from the vehicle's current location and ending at the first waypoint, *FindPath* generates *ArcVote* events at a fixed frequency attempting to track the planned path. The path planner is not deterministic in the time it takes to plan a path. If a path can not be planned in the time available, a vetoed *ArcVote* event is published. If the vehicle is unable to track the planned path because of obstacles encountered as the vehicle moves forward, *FindPath* is forced to re-plan.

Item	Type	Description
Veto	bool	Setting to true vetoes this arc
Vote	float	A number between 0 and 1 indicating the desirability of the arc
Certainty	float	A number between 0 and 1 indicating the behaviour's belief in its data
MaxSpeed	float	The maximum speed at which the behaviour finds it acceptable for the vehicle to travel this arc

Table 23: A vote for a single candidate arc

The run-time operation of *FindPath* is configured by parameters defined in the Raptor's XML configuration file. These four parameters, located under the *FindPath* section, include the ChannelName, MaxLookahead, MinLookahead and TrackingTolerance. Table A.9 lists these parameters and provides an explanation of the significance and usage.

3.3.2 Obstacle Avoidance Component

The *Obstacle Avoidance* component subscribes to *Traverse* events, and evaluates a set of candidate arcs for their suitability. A typical traversability map and the candidate arcs are shown in Figure 10. The black arcs shown in the picture have been vetoed because of the discrete obstacles in their path, while the blue arcs have been deemed safe to travel.

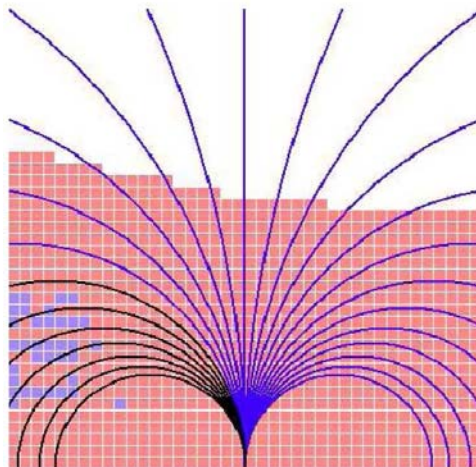


Figure 10: A Traversability Map with overlaid candidate arcs

The *Obstacle Avoidance* component's operation is determined by a number of run-time parameters. These parameters are located under the *ObsAvoid* section of the Raptor's XML configuration file. Table A.10 provides a complete list of these parameters and describes their significance and usage.

3.3.3 Pure Pursuit Component

The *Pure Pursuit* component allows the Raptor UGV to seek high level goal GPS waypoints provided by the human controller, by attempting to follow the straight line segments between them. The algorithm continually calculates the candidate arc necessary to return the vehicle to the path, at a specified look-ahead distance. This results in smooth path following behaviour, as is illustrated in Figure 11.

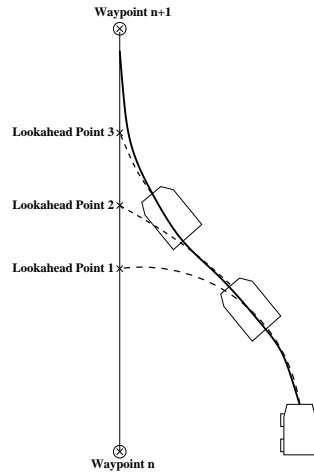


Figure 11: Illustration of the method *Pure Pursuit* uses to follow paths.

The waypoints given to the vehicle from the control station are passed to the *Pure Pursuit* component via its polled interface defined in the *VehiclePlan* interface, shown in Table 24. The *WaypointGroupIDLs*, referred to in the table, consist of arrays of latitude/longitude pairs, which define the sequence of waypoints to follow. The *PatrolMode*, defined in the table, indicates whether or not the vehicle should continue to cycle through those waypoints after it has completed the original sequence..

Poll Interface	Description
setWaypointList(WaypointGroupIDL)	Assigns waypoints to the vehicle
WaypointGroupIDL getWaypointList()	Retrieves the vehicles current working waypoint list
WaypointIDL getCurrentWaypoint()	Retrieves the waypoint that the vehicle is trying to reach
setPatrolMode(boolean)	Tells the vehicle to loop through the waypoints, or not
boolean getWaypointList()	Reports whether or not the vehicle is in Patrol Mode
setHalt(boolean)	Stops the vehicle if set to true (i.e. veto all arcs)

Table 24: Polled Interface for Waypoints in *Pure Pursuit*

In order to operate in concert with other behaviours, the *Pure Pursuit* module does not directly output a vote for the ideal arc. Rather, it spreads the votes out on a Gaussian distribution around the ideal arc, so that the decision making process can select candidate arcs that will avoid obstacles, while still providing goal directed behaviour. A typical vote set scaling output from the *Pure Pursuit* component is shown in Figure 12. This algorithm has no configurable parameters.

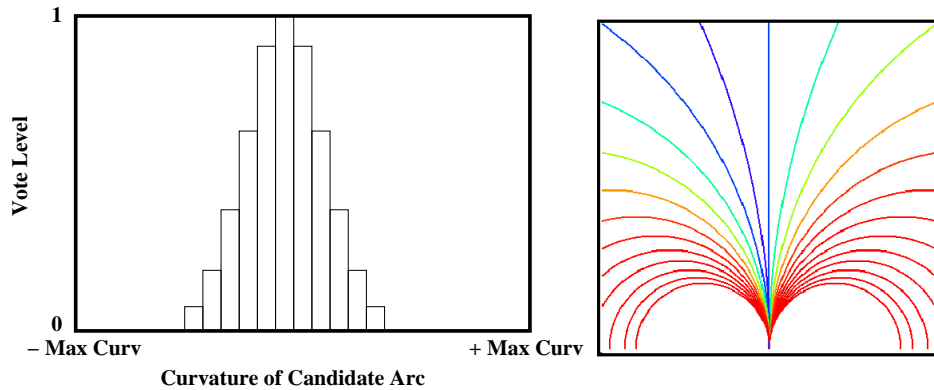


Figure 12: A selection of candidate arcs, coloured to indicate those more desirable to follow the intended path.

3.3.4 Obstacle Detection Component

Drawing from one or more range sensors, the simple *Obstacle Detection* component halts the vehicle if it finds any positive or negative obstacles in the vehicle’s immediate path. This functions as a safety system that prevents the vehicle from damaging itself. The area in front of the vehicle to be checked, as well as the acceptable height and depth of obstacle, are user-definable via the parameters defined in the Raptor’s configuration file. Table A.11 lists these parameters and can be found in the appendix.

The algorithm functions by checking the 3D range readings from the *Nodding Laser* component, adjusting for the height at which the sensor is mounted above the wheels. If it finds a user-specified number of range returns greater than a certain height tolerance above the ground plane, or lower than a certain depth below, it will halt the vehicle via an *ArcVote* event.

3.4 Decision Making

The *ArcArbiter* is the sole decision making component under the current Raptor configuration. It asynchronously receives votes from each “voting” component shown in Figure 7 and uses these votes to make decisions. With each vote event, the *ArcArbiter* component re-evaluates its decision and is therefore, highly reactive to incoming data. The *ArcArbiter* component commands the vehicle actions by passing *steering* and *velocity* commands to the *Vehicle Control* component via the polled *setVelocity* interface. Table 25 provides details about this component’s implementation.

Component	Design Pattern	Subscribed Events	Published Events	Update Rate	Polling
Arc Arbiter	SPS	ArcVote	None	Upon event	setVelocity

Table 25: Decision Making Component

As is the case with other Raptor components, the *ArcArbiter*'s run-time operation is influenced by parameters, located under the *ArcArbiter* section, in the Raptor's XML configuration file. A complete listing of these parameters, including explanations of their significance, is found in Table A.12.

3.5 Vehicle Control

Physical control of the Raptor UGV, including steering angles, vehicle speed and braking, is achieved by the on-board MPC555 microcontroller. The low level MPC555 microcontroller communicates with the high level *Vehicle Control* component via a serial port. Data flows bi-directionally over this communications channel with MPC555 providing status reports at fixed intervals, while the *Vehicle Control* component sends control commands as required. The key control command sent is *setVelocity*, which defines both the rotational and translation velocity for the Raptor. Other control commands are also available and are shown in Table 26.

Poll Interface	Polled Object	Implementation
setVelocity()	VelocityIDL	Sets Raptor velocity
getTargetVelocity()	none	Return target velocity
startVehicle()	none	Start the vehicle
stopVehicle()	none	Stop the vehicle
setJoystickVelocity()	VelocityIDL	Set the Velocity using a joystick
setArbiter()	none	Turn Arbiter on/off
getMinMaxVelocity()	none	Returns Min/Max velocities

Table 26: Vehicle Control Polling Interfaces

Given that the *Vehicle Control* component interfaced with MPC555 microcontroller using a serial port, it was derived from the most suitable design pattern - the *Publish Server and Reactor*. This design pattern features both an ACE reactor and a Miro server; thus, the component can process serial data using the ACE reactor callback function, while the Miro server concurrently publishes events and responds to poll requests. This component publishes a single event type, the *VelocityData*, that provides information with respect to the vehicle's current speed in mm/sec, as shown in Table 27.

The *RaptorCommandConfig* section, of the Raptor's XML configuration file, influences the *Vehicle Control* component's run-time operation. Table A.13, located in the appendix,

Component	Design Pat.	Subscribed Events	Published Events	Update Rate	Polling
Vehicle Cont.	PSR	None	VelocityData	Variable	none

Table 27: *Vehicle Control Component*

provides a complete list of configurable parameters and describes each parameter's purpose.

4 Utilities

DRDC developed a series of graphical and text based utilities that assist the researcher in visualizing and understanding the performance of the Raptor's various components. The following sections describe the utilities that were developed for each Raptor component.

4.1 Range Sensors

Two distinct graphical interfaces have been developed to display the raw range data acquired by the ranging sensors. The *QtRange3dSensor* utility graphically displays the range data produced by the nodding SICK laser. The stereo vision point clouds are displayed using the *qtEventStereo* utility. The text based utilities *Sensor3dStream* and *Sensor3dGet* simply display the range data as ASCII text.

4.1.1 QtRange3dSensor

The *QtRange3dSensor* utility, shown in Figure 13, displays the raw range data acquired by the nodding SICK laser. Before displaying, the data must be transformed from the Laser frame into the Raptor frame, thus the *ModelServer* component must be queried for the appropriate transform (see Table 30). Once this transform has been acquired, the *QtRange3dSensor* utility periodically polls the *Nodding Laser* component, transforms the acquired raw range data into the Raptor frame and displays the results⁴. Table 28 shows the objects and interfaces required by this utility.

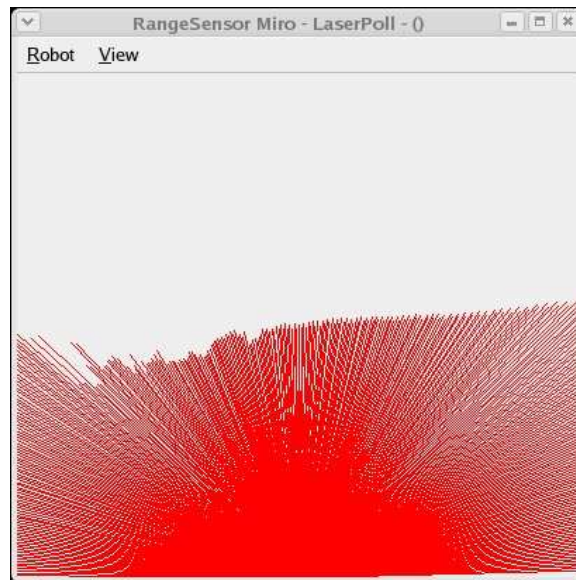


Figure 13: Graphical Range Display

⁴The results may be displays a X distances or Z elevations.

Object Server	Resolution Variable	Polled Interface	Polled Object
ModelServer	Platform_var	getTransformation()	PoseTransformIDL_var
Laser	Range3dSensor_var	get3dLaserFullScan()	Range3dLaserEventIDL

Table 28: *QtRange3dSensor Objects and Interfaces*

The polled implementation means this utility can only display range data that is produced by the *Nodding Laser* component or a component that support polling; logged data replayed via the *LogPlayer* utility can not be directly display⁵. If it is necessary to graphically display logged laser range data the *QtRange3dServer* server acquires *Laser* events, while providing the required polling interface.

4.1.2 Sensor3dStream

The *Sensor3dStream* utility is a text based methods for printing range data. *Sensor3dStream* subscribes to the events listed in Table 29. It polls the *ModelServer* component using the object shown in Table 30, then transforms the data to the Raptor frame and, finally, the range data is printed in a text format.

Event Channel	Event Name	Event Object
EventChannel	Laser11	Range3dLaserIDL/Range3dSeqEventIDL
EventChannel	Laser12	Range3dLaserIDL/Range3dSeqEventIDL
EventChannel	Stereo	Range3dStereoEventIDL/Range3dArrayEventIDL

Table 29: *Sensor3dStream Subscribed Events*

Object Server	Resolution Variable	Polled Interface	Polled Object
ModelServer	Platform_var	getTransformation()	PoseTransformIDL_var

Table 30: *Sensor3dStream Polled Objects from the Model Server*

4.1.3 Sensor3dGet

The *Sensor3dGet* utility polls the *Nodding Laser* or *Stereo Vision* component for the objects listed in Table 31.

The range data must be transformed into the Raptor frame; thus it also polls the *Model Server* component for transform objects, transforms the range data and prints in a text format.

⁵The LogPlayer doesn't support polling.

Object Server	Resolution Variable	Polled Interface	Polled Object
NodLaser	Range3dSensor_var	get3dSeqFullScan()	Range3dSeqEventIDL
NodLaser	Range3dSensor_var	get3dLaserFullScan()	Range3dLaserEventIDL
NodLaser	Range3dSensor_var	get3dArrayFullScan()	Range3dArrayEventIDL
NodLaser	Range3dSensor_var	get3dStereoFullScan()	Range3dStereoEventIDL
ModelServer	Platform_var	getTransformation()	PoseTransformIDL_var

Table 31: *Sensor3dGet Polling Variables and Objects*



Figure 14: *QtBodyViewer Display*

4.2 QtBodyViewer

The *QtBodyViewer* utility is a simple Qt application designed to help visualize the layout of frames in the ModelServer Geometry files. *QtBodyViewer* loads and presents all frames in a Qt 3D environment (see Figure 14). By using the mouse, the frame layout may be inspected from any angle and can be interrogated for their body-frame specification.

Note that both the *QtBodyViewer* and *QtEventStereo* utilities rely on the third party Qt library: *libQGLViewer*.

4.3 QtEventPose

To inspect ModelServer's event driven output, a simple Qt utility, *qtPoseEvent* has been provided (see Figure 15). Event based Miro applications pose some minor problems for Qt, specifically IDL types do not cross the Qt-Miro boundary well. Typically Qt IDL pseudotypes are required to bridge the gap between Qt and Miro. *QtEventPose* simply provides a GUI window onto PoseTransformIDL events (see Table 21) as they are published on the EventChannel.

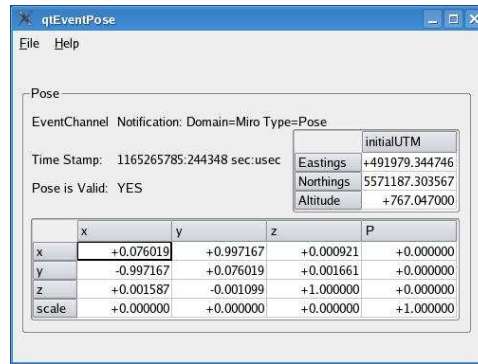


Figure 15: *QtEventPose Display*

4.3.1 QtEventStereo

The *QtEventStereo* utility is a simple Qt application designed to help visualize the Digiclops point cloud within ModelServer coordinate system. Similar to *QtBodyViewer*, *qtEventStereo* loads and presents all frames in a Qt 3D environment (see Figure 16) as well as the Digiclops point cloud. By using the mouse, the frame and point cloud layout may be inspected from any angle and frames can be interrogated for their body-frame specification.

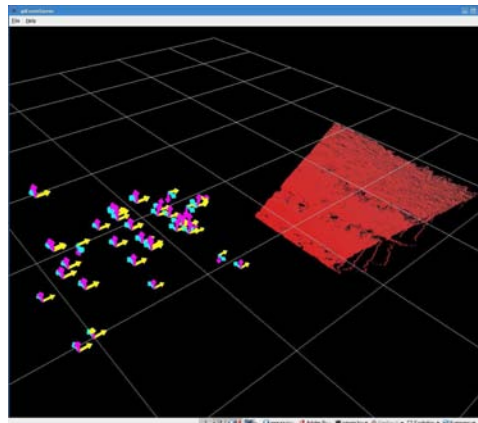


Figure 16: *QtEventStereo Display*

4.4 Maps

Graphical displays are an invaluable interfaces assisting in the development, understanding and monitoring of maps and are key tools in understanding the decision making process as well. The *QtMap* interface graphically displays global terrain maps and traversability maps. Egocentric terrain and traversability maps are displayed using the *LocalMap* utility, which can also display the candidate arcs created by the *ArcArbiter* component. Each map type is discussed, in further detail, in the following sections.

4.4.1 QtMap

The global terrain map is a $2\frac{1}{2}$ -D terrain map is centred on the vehicle and is referenced to the cardinal directions (N,E,S,W). Thus, in the global terrain map shown in Figure 17, North points towards the top of the map.

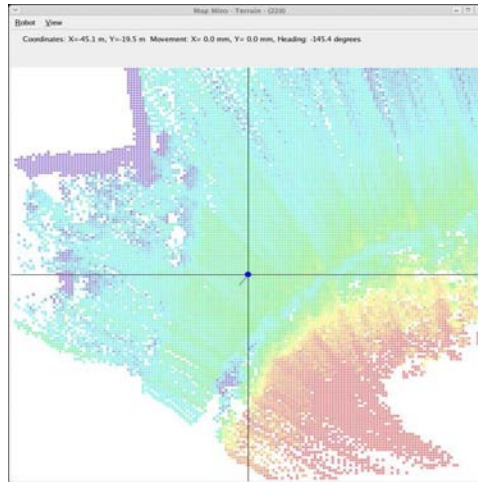


Figure 17: Typical Global Terrain Map

The global terrain map acquires data via a polling process using the appropriate interface, where the data is supplied from the *Terrain Map* component. Table 32 shows the object polled by this utility⁶.

Object Server	Resolution Variable	Polled Interface	Polled Object
Terrain Map	Map_var	getMapArray()	MapArrayEventIDL
Terrain Map	Map_var	getMapSeq()	MapSeqEventIDL

Table 32: Global Terrain Map Interfaces and Polled Objects

4.4.2 LocalMap

The local map displays an egocentric world, which corresponds to view from the front bumper of the vehicle. Figure 18 shows a typical the local map, where the vehicle's bumper is located at the bottom of the image.

The local map also acquires data via a polling process and once again the data is supplied from the *Terrain Map* component. Table 34 shows the object(s) polled by this utility and interfaces required to acquire the object(s).

⁶This utility can display either array or sequence based maps; thus, the table shows both interfaces and objects.

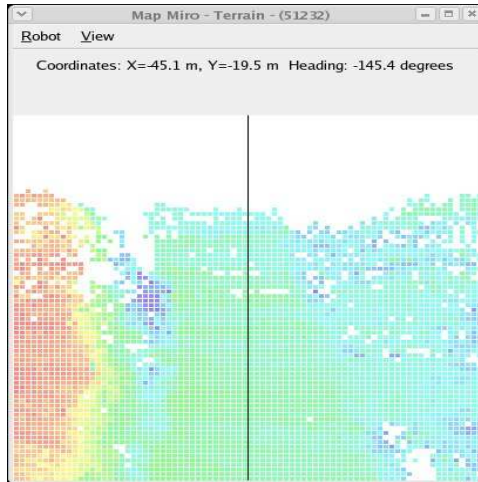


Figure 18: Typical Egocentric Local Map

Object Server	Resolution Variable	Polled Interface	Polled Object
Terrain Map	Map_var	getEgoMapArray()	EgoMapArrayEventIDL
Terrain Map	Map_var	getEgoSeqArray()	EgoMapSeqEventIDL

Table 33: Egocentric Terrain Map Interfaces and Polled Objects

4.4.3 Launcher

As depicted in Figure 19, this utility displays the state of running components and permits users to run, kill and restart them on the fly. It uses an XML parameter file in which paths, command names, and command line parameters are specified. Though not a Miro component per se, *launcher* can be used to enable/disable a process, apply variable delays between process starts, launch a process in a terminal (TTY), and launch multiple processes automatically. *Launcher* temporarily fills a role for component monitoring and control.

Runtime: On startup *launcher* loads the XML file, kills off any preexisting opening processes listed in the file and awaits instructions. New XML profiles may be opened at any time, killings all running processes prior to the new file. *Launcher* does not discriminate between child processes, and will kill off any preexisting processes with the same name.

Enabled Processes: Processes can be 'enabled' as part of the autolaunch group (the default XML setting) or disabled through the GUI at any time. Any process in the autolaunch group can be launched or killed with a single keystroke (CTRL+SHIFT-L and CTRL-K respectively).

Terminals: When launched, a component will be given an output terminal if TTY is set to TRUE. When a process TTY is deliberately or spontaneously killed, the TTY will disappear. The reverse is not true, closing a TTY will not kill the process. Unchecking TTY in launcher does not close or kill a TTY.

	Name	Enable	TTY	dT(s)	State	PID	%CPU	Mem(KB)
1	Naming_Service	<input checked="" type="checkbox"/>	<input type="checkbox"/>	3	ready	9909	0	732
2	nsview	<input checked="" type="checkbox"/>	<input type="checkbox"/>	3	killed	0	0	0
3	eventBase	<input checked="" type="checkbox"/>	<input type="checkbox"/>	3	killed	0	0	0
4	LogNotify	<input type="checkbox"/>	<input type="checkbox"/>	3	killed	0	0	0
5	LogPlayer	<input type="checkbox"/>	<input type="checkbox"/>	4	ready	9915	0.2	46544
6	RaptorService	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
7	3dmgService	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
8	GpsSokkiaService	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	5	killed	0	0	0
9	NodLaserService	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
10	modelServer	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1	ready	10022	0.3	53048
11	LaserSafetyService	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
12	HealthSafetyService	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
13	TerrainService	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1	killed	0	0	0
14	TraverseService	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1	killed	0	0	0
15	purePursuit	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
16	obsAvoid	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
17	arcArbiter	<input type="checkbox"/>	<input checked="" type="checkbox"/>	3	killed	0	0	0
18	DpPanTilt	<input type="checkbox"/>	<input checked="" type="checkbox"/>	7	killed	0	0	0
19	qtEventImu	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	killed	0	0	0
20	qtEventPose	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	ready	9957	0.2	12664
21	QtMap	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	killed	0	0	0
22	qtHealthSafety	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	killed	0	0	0
23	qtEventVehicle	<input type="checkbox"/>	<input type="checkbox"/>	7	killed	0	0	0
24	qtEventStereo	<input type="checkbox"/>	<input type="checkbox"/>	7	killed	0	0	0

Figure 19: Launcher Display

GUI Interaction: Processes can be selected with a mouse LEFT-CLICK. A group of processes can be assembled through standard SHIFT or CTRL mouse selection.

Launching Processes: Selected components may be launched through CTRL-L, while autolaunching is invoked through CTRL-SHIFT-L. Similarly, selected components may be killed through CTRL-K.

Launch Delay: After each process launch, Launcher will wait dT seconds before launching another component. “dT” may be set in the XML file or through the GUI.

Killing Processes: Processes are killed through a “killall -9 *commandname1 commandname2*” call. *Killing processes does not guarantee that peripheral devices will stop what they are doing.* – only the kill switch can do that.

Display: The displayed values PID, %CPU, and Memory are gathered through a system call: ps ocomm= -opid= -opcpu= -osize=.

To discriminate between xhosted sessions, the hostname of the computer is displayed in the status bar at the lower left corner of the launch window. Other messages will appear on the status bar periodically. The display is refreshed at a rate set in the XML file (RefreshRate, the default is 5 seconds) that can be set through the VIEW menu.

Parameter	Description	Type(Default)
Path	the Component's Path	string(N/A)
Name	the Component Name	string(N/A)
Enabled	Component is in autolaunch group	bool(false)
InTTY	Component needs terminal	bool(true)
Period	Delay next Component launch (in seconds)	int(3)
Arguments	Component's command line arguments	string vector(N/A)

Table 34: *Launcher Process XML File entries. Launcher supports environment variables (e.g. \$NSC, \$NS, \$TAO_ROOT, etc.) in both Path and Argument fields.*

5 Effort

Implementing the autonomous capabilities for the Raptor UGV, under the “Architecture for Autonomy“, required a significant research and development effort. The SLOCCount application was used to summarize and quantify this effort. SLOCCount (pronounced ”sloc-count”) is a suite of programs for counting physical source lines of code (SLOC) in potentially large software systems. Thus, SLOCCount is a ”software metrics tool” or ”software measurement tool” [27]. Each directory under the drdcMiro root was analyzed by the SLOCCount application. The effort is broken into four separate categories:

- Configuration and Testing software.
- The core components that implement autonomous capabilities.
- Text based utilities for printing the status of the system’s components.
- Utilities that are useful to visualize and display the status of the system’s components.

Table 35 shows the effort applied to the configuration and testing process.

Component	Predominate Language	Lines of Code	Effort Person-Years
Config	sh	7783	1.72
Tests	cpp	241	0.04
Total Effort		8024	1.76

Table 35: *Estimated Effort associated with Configuration and Testing*

Text based tools are useful in presenting certain types of status information. The effort required to develop these tools is detailed in Table 36.

As is shown in Table 37, the majority of the software effort resided rested with development of components; thus, with the implementation of autonomous capabilities.

Although text based presentations are useful under certain circumstances, under other circumstances a graphical interface is superior. Table 38 shows the effort expended to develop graphical interfaces and, thus, the visual presentation of data.

Overall, researchers developed a total of 60,800 source lines of code that corresponds to an estimated 13.01 person-years of development effort.

Component	Predominate Language	Lines of Code	Effort Person-Years
3dmg	cpp	452	0.09
arcVote	cpp	101	0.02
Examples	cpp	498	0.10
gpsSokkia	cpp	176	0.03
map	cpp	451	0.09
missionPlan	cpp	133	0.02
NodLaser	cpp	53	0.01
panTilt	cpp	38	0.01
pioneer	cpp	75	0.01
platform	cpp	680	0.13
range3dSensor	cpp	431	0.08
raptor	cpp	205	0.04
vehiclePlan	cpp	44	0.01
Total Effort		3337	0.71

Table 36: *Estimated Effort to Implement Text based Utilities*

Component	Predominate Language	Lines of Code	Effort Person-Years
3dmg	cpp	1087	0.22
arcArbiter	cpp	649	0.13
controlStation	cpp	840	0.17
digiclops	cpp	506	0.10
DIIPanTilt	cpp	696	0.14
eventChannel	cpp	216	0.04
findPath	cpp	5257	1.14
globalMap	cpp	4207	0.90
gpsGarmin	cpp	914	0.18
gpsSokkia	cpp	1253	0.25
healthSafety	cpp	1174	0.24
laserSafety	cpp	422	0.08
map_terrain	cpp	2180	0.45
map_traverse	cpp	1899	0.39
miro	cpp	7650	2.07
nodlaser	cpp	1869	0.39
obsAvoid	cpp	687	0.13
platform	cpp	1896	0.39
purePursuit	cpp	749	0.15
raptorCommand	cpp	2631	0.55
urgLaser	cpp	670	0.13
vehicleIntel	cpp	394	0.08
waypointArbiter	cpp	156	0.03
Total Effort		38034	7.96

Table 37: *Estimated Core Component Efforts*

Component	Predominate Language	Lines of Code	Effort Person-Years
launcher	cpp	920	0.18
localMap	cpp	870	0.17
nsview	cpp	403	0.08
PanTilt	cpp	239	0.04
PioneerJoy	cpp	131	0.02
qtBodyViewer	cpp	669	0.13
qtEventImu	cpp	442	0.08
qtEventPlanMap	cpp	1570	0.32
qtEventPose	cpp	636	0.12
qtEventStereo	cpp	938	0.19
qtHealthSafety	cpp	745	0.15
qtMap	cpp	750	0.15
qtPollDIIPanTilt	cpp	617	0.12
qtRaptor	cpp	1252	0.25
rangeSensor	cpp	557	0.11
rapJoy	cpp	136	0.02
raptorGUI	cpp	324	0.06
widgets	cpp	447	0.09
Total Effort		11405	2.58

Table 38: *Estimated Effort to Implement Graphical based Utilities*

6 Conclusions

DRDC required a modular, extensible, flexible and scalable framework to support its unmanned vehicle program. Adopting a Component Based Software Engineering philosophy and building upon the Miro framework resulted in the “Architecture for Autonomy” . The AFA has deep open source roots. Linux is the preferred operating system; configuration, compiling and linking uses the GNU toolchain; open source libraries are extensively used; ACE, an open source communications middleware toolkit, and the TAO implementation of CORBA underlie the Miro framework. Subsequently, the AFA and all of its prerequisites are freely available; there are no licensing issues to restrict its usage and dissemination.

Under the AFA researchers developed a suite of components, that when operating in unison, allowed the Raptor UGV to exhibit autonomous traits. This component-based implementation, relying on the CORBA’s network transparency capabilities, delivered the modularity, extensibility, flexibility and scalability that both researchers and the program required. The flexibility provided by the ACE/TAO middleware toolkits allowed the Raptor’s software architecture emerged to meet the system requirements. The Raptor UGV components were divided into six logical domains: sensing; information processing and representation; planning and goal seeking; decision making; vehicle control and utilities. The sensing domain included five components; information processing and representation was comprised of four components; planning and goal seeking encompassed another four components; single components implemented decision making and the vehicle control; and there were nine utility components. Data transparently flowed between the various components via the asynchronous delivery of events, or through a polling mechanism. Regardless of the delivery mechanism, all data transfers were defined by standard interfaces. These interfaces were developed under the Interface Definition Language and, thus, are portable across networks and operating systems.

From a developmental perspective over 60,000 source lines of code, representing an effort of 13 person-years, were required to implement the Raptor UGV. The development of core autonomous capability consumed a majority of the effort; requiring approximately 8 person-years to implement. The implementation of graphical and text based utilities were next major consumer of resources, and required approximately 3.25 person-years to complete. The remainder of the effort was expended on the system setup, configuration and testing.

The AFA underlies the autonomy software developed for the Raptor UGV, but its flexibility allows it to be easily adapted to other unmanned vehicle platform. Work is currently underway to extend the Miro framework to support the wheeled Pioneer platforms. Additionally, researchers at Valcartier are extending Miro framework to support unmanned air vehicles, namely an autonomous helicopter.

This page intentionally left blank.

Annex A: Tables of Configuration Parameters

Parameter	Default	Description
EventChannelName	EventChannel	The event channel's name
EventName	Laser11	Name of events published
LaserID	11	I.D. encoded in the event structure
Host	131.135.74.200	Nodding Laser I.P. Address
Port	24750	Nodding Laser Port Number
Baudrate	500000	Nodding Laser Baudrate
Motorrate	15	Nodding Rate Deg/sec
LowAngle	-40	Lowest Nod angle, degrees
HighAngle	-10	Highest Nod angle, degrees
LaserOffsetAngle	-2	A tweaking parameter to adjust the mount angle
Laserres	50	Laser Resolution, see the SICK Manual
MountAngle	-14	Mount angle of the laser - Used for adaptive nodding
MountHeight	3.0	Mounting Ht. above ground (m) - Used for adaptive nodding
ScanDistance	0.2	Scan distance (m) - Used for adaptive nodding

Table A.1: *Nodding Laser Component's Configuration Parameters*

Parameter	Default	Description
EventChannelName	EventChannel	The event channel's name
EventName	Stereo	Name of event published
MaxRange	8000	Maximum depth range in mm
frameRate	2	The Event Publication rate in frames/sec.

Table A.2: *Stereo Vision Component's Configuration Parameters*

Parameter	Default	Description
Device	/dev/ttyUSB0	The serial port where the hardware is found
ChannelName	EventChannel	Name of event channel to publish data
Notify	true	Toggles whether or not the component publishes its data
EventName	Imu	The name of the event on the Event Channel.

Table A.3: *Imu Component's Configuration Parameters*

Parameter	Default	Description
Device	/dev/ttyUSB0	The serial port where the hardware is found
ChannelName	EventChannel	Name of event channel to publish data
Notify	true	Toggles whether or not the component publishes its data
EventName	Gps	The name of the event on the Event Channel.

Table A.4: *GPS Sokkia Component's Configuration Parameters*

Parameter	Default	Description
Device	/dev/ttyS0	Serial port
LfdParam	N/A	Left wheel distance calibration
RfdParam	N/A	Right wheel distance calibration
SteerParam	N/A	Steering calibration parameters
SpeedRepTime	1000	Speed report interval (ms)
WheelDistRepTime	200	Wheel distance report interval (ms)
RaptorOdometryEvent	RaptorOdometry	Name of published Events

Table A.5: *Wheel Odometry Component's Configuration Parameters*

Parameter	Default	Description
EventChannelName	EventChannel	The event channel's name
EventName	Terrain	Name of event published
Depth	32000	Map depth in mm
Width	32000	Map width in mm
Gridsize	200	Map Grid size in mm
notify	true	Published events enabled, True/False
RangeEventName1	Laser11	Subscribed Range Event
RangeEventName2	Laser12	Subscribed Range Event
RangeEventName3	Stereo	Subscribed Range Event
PoseEventName	Pose	Subscribed Pose Event
MapEventInterval	0.25	Create map events on a timed interval, sec.
MapUpdateTime	7.0	Time required to fill map with data, sec.
PlatformEvent	ModelService	Polled Object for geometry
GroundFrame	Raptor: Front-BumperCenter	Map Frame
SickLaserFrame1	FrontNoddingSICKLeft: AxleCenterFreeEnd	Left SICK Laser Frame
SickLaser1_ID	11	Left Laser I.D.
SickLaserFrame2	FrontNoddingSICKRight: AxleCenterFreeEnd	Right SICK Laser Frame
SickLaser2_ID	12	Right Laser I.D.
DigiclopsFrame	Digiclops:ImagePlane	Digiclop Frame
BumperHeight	-500	Ground to bumper Ht., mm
MaxVelocity	6000	Max vehicle velocity, mm/sec.
PoseInterval	0.1	Pose update interval, sec. Used with MaxVelocity to audit position changes
OrientationDevError	2.0	Estimate Orientation Error, deg.
StereoPeriod	0.5	Stereo Event Rate, sec. Used to equal Stereo and Laser data densities

Table A.6: *Terrain Map Component's Configuration Parameters*

Parameter	Default	Description
StepOn	1	Turn the Step Hazard calculation on/off
SlopeOn	1	Turn the Slope Hazard calculation on/off
StepPercent	0	Calculate Step Hazard as a percentage
Stepheight	300	Step hazard height
Maxslope	20	Slope hazard angle
ChannelName	EventChannel	EventChannel name
GlobalMap	true	Turn the Global Traversability Map on/off
EgoMap	true	Turn the Ego Traversability Map on/off
Farfield	8000	Distance (mm) of the farzone
Farfieldscale	1.5	Step hazard scaling factor for the farfield area
Farfieldlopescale	1.5	Slope hazard scaling factor for the far-field area
MapEventInterval	1	Rate (seconds) at which events are produces
MapUpdateTime	7	Delay (seconds) between events after a clear map has been issued
GlobalTravMapDescription Depth Width Gridsize EventName	32000 32000 500 Traverse	Description of the Global Traversability Map Depth (mm) of map Width (mm) of map Size of map cells (mm) Event name
EgoTravMapDescription Depth Width Gridsize EventName	16000 16000 500 Traverse	Description of the Ego Traversability Map Depth (mm) of map Width (mm) of map Size of map cells (mm) Event name
TerrainMapDescription Depth Width Gridsize EventName	32000 32000 200 Terrain	Description of the Global Terrain Map events subscribed to Depth (mm) of map Width (mm) of map Size of map cells (mm) Event name
EgoTerrainMapDescription Depth Width Gridsize EventName	16000 16000 200 Terrain	Description of the Ego Terrain Map events subscribed to Depth (mm) of map Width (mm) of map Size of map cells (mm) Event name

Table A.7: *Traversability Map Component's Configuration Parameters*

Parameter	Default	Description
EventChannelDescription		
Subscriptions	Gps Gps Imu Imu	IDL-type EventName IDL-type EventName
Publications	Pose Pose	IDL-type EventName
ResolvedNames	Imu Imu	IDL-type EventName
OfferedNames	Platform ModelService	IDL-type EventName
UseRawPose	true	don't use filtering
ModelFileName	050917raptor01.XML	name of geometry model file
UpdatePeriod	0	0: event driven, > 0: timed updates
FidgetDelay	20	delay before fidgeting
FidgetPeriod	300	period between fidgets

Table A.8: *ModelServer's Configuration Parameters*

Parameter	Default	Description
MaxLookahead	20.0	The maximum distance (in planning grid units) along the planned path from the point on the path closest to the vehicle's current position that the tracking algorithm will search.
MinLookahead	7.0	The minimum distance (in planning grid units) along the planned path from the point on the path closest to the vehicle's current position that the tracking algorithm will search.
TrackingTolerance	30.0	The maximum distance (in planning grid units) the vehicle can be from the planned path without forcing a replanning episode.
PlanUpdateDelayTime	5.0	The delay (s) prior to the first call to generate a ArcVoteIDL event.
PlanEventInterval	1.0	The interval (s) between successive ArcVoteIDL events.

Table A.9: *FindPath Component's Configuration Parameters*

Parameter	Default	Description
CurveMakingConst	0.01	A distance(m) which the algorithm steps through to construct an approximation of the candidate arc
VehicleRadius	0.5	A radius(m) of a circle which approximates the size of the vehicle
MaxVelocity	2.0	The top speed(m/s) which the algorithm will allow for any arc
MinVelocity	0.5	The lowest speed(m/s) the algorithm will suggest for any arc before vetoing it
NearZone	0	The number of map rows immediately in front of the vehicle to ignore obstacles. Makes it possible to ignore parts of the map.
FarZone	39	The number of map rows immediately in front of the vehicle to evaluate. Makes it possible to ignore parts of the map which are too far away to be important.
DistFactor	0	A percentage by which to discount the cells estimation for distance
MinDistDiscount	1	A number between 0 and 1 which indicates how low we can discount for distance (1 = no discount)
MapDepth	16000	The depth of the Traversability Map(mm).
MapWidth	16000	The width of the Traversability Map(mm).
MapGridsize	0.01	The size of each cell in the Traversability Map(mm).
DiscountSpeeds	false	Whether or not the vehicles speed will be controlled based on the obstacle map.
CostUnknownCells	false	Whether or not to consider unknown portions of the map as obstacles.
VetoUnknownCells	false	Whether or not to consider unknown portions of the map as obstacles.

Table A.10: *Obstacle Avoidance Component's Configuration Parameters*

Parameter	Default	Description
SafetyHeight	0.3	The maximum obstacle height considered safe(m)
SafetyDepth	0.3	The maximum obstacle depth considered safe(m)
SafetyDist	3	How far in front of the vehicle to check for obstacles(m)
VehicleWidth	1	The approximate width of the vehicle(m)
DefaultYMax	3	How far to the left/right of the vehicle to check for obstacles(m)
LaserHeight	2	The height of the laser sensor above the ground
NumHitsTolerance	5	How many laser range returns outside the safety height/depth triggers an obstacle detection
MaxLaserRange	8191	How far the laser sensor can see (mm)
CheckPosObs	true	Whether or not to look for positive obstacles
CheckNegObs	true	Whether or not to look for negative obstacles
ResetCounter	200	How many laser scans to wait after a halt before rechecking for obstacles

Table A.11: *Obstacle Detection Component's Configuration Parameters*

Parameter	Default	Description
ObsWeight	1	The importance of Obstacle Avoidance's votes (1 being standard)
PursuitWeight	1	The importance of Pure Pursuit's votes
PlannerWeight	1	The importance of Find Path's votes
LaserSafetyWeight	1	The importance of Obstacle Detections's votes
TransVelocitySetPoint	0.5	The maximum allowed vehicle speed (m/s)
StaleVoteTimeSec	1	How long to hang on to a vote before considering it stale
StaleVoteTimeUSec	500000	How long to hang on to a vote before considering it stale

Table A.12: *Arc Arbiter Component's Configuration Parameters*

Parameter	Default	Description
RaptorMotionEvent	RaptorMotion	Event name to publish under
Device	/dev/ttyS0	Serial device to write to
MaxSteeringAngle	30	Maximum allowable steering angle (degrees)
MaxRotVelocity	1	Maximum allowable rotational velocity (Rad/s)
MaxTransVelocity	2000	Maximum allowable translational velocity (mm/s)
RfdParam	N/A	Right wheel calibration parameter
LfdParam	N/A	Left wheel calibration parameter
PedalParam	N/A	Pedal calibration parameter
SteerParam	N/A	Steering calibration parameter
SteerPID	N/A	Steering PID parameters

Table A.13: *Vehicle Control Component's Configuration Parameters*

References

- [1] Broten, G., Erickson, D., Giesbrecht, J., Monckton, S., and Verret, S. (2003), Engineering Review of ANCAEUS/AVATAR An Enabling Technology for the Autonomous Land Systems Program?, (DRDC Suffield TR 2003-167) Defence R&D Canada – Suffield.
- [2] Broten, G., Monckton, S., Giesbrecht, J., Verret, S., Collier, J., and Digney, B. (2004), Towards Distributed Intelligence - A high level definition, (DRDC Suffield TR 2004-287) Defence R&D Canada – Suffield.
- [3] Broten, G., Verret, S., and Digney, B. (2004), Unmanned Ground Vehicle Software Development Environment, (DRDC Suffield TM 2004-060) Defence R&D Canada – Suffield.
- [4] Gerkey, Brian, Vaughan, Richard T., and Howard, Andrew (2003), The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323.
- [5] Roy, M. Montemerlo N. and Thrun, S. (2003), Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit, In *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*.
- [6] Cote, C., Letourneau, D., Valin, F. Michaud and J-M., Brosseau, Y., Raievsky, C., Lemay, M., and Tran, V. (2004), Code Reusability Tools for Programming Mobile Robots, In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.
- [7] Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G. (2002), Miro - Middleware for Mobile Robot Applications, *IEEE Transactions on Robotics and Automation*.
- [8] Enderle, S., Utz, H., Sablatnog, S., Simon, S., Kraetzschmar, G., and Palm, G. (2001), Miro - Middleware for Autonomous Mobile Robots, *International Federation of Automatic Control*.
- [9] Brooks, A., Kaupp, T., Makarenko, A., Oreback, A., and Williams, Stefan (2005), Towards Component-Based Robotics, In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
- [10] Broten, G., Monckton, S., Collier, J., and Giesbrecht, J. (2006), Architecture for Autonomy, In Grant R. Gerhart, Douglas W. Gage, Charles M. Shoemaker, (Ed.), *Proceedings of SPIE, Unmanned Vehicle Research At DRDC*, Vol. 6230, Orlando, FL.
- [11] Broten, G. and Monckton, S. (2005), Frameworks and Middleware for Unmanned Ground Vehicles, In Grant R. Gerhart, Douglas W. Gage, Charles M. Shoemaker, (Ed.), *Proceedings of SPIE, Unmanned Ground Vehicle Technology VII*, Vol. 5804, pp. 655–664, Orlando, FL.

- [12] Huston, S., Johnson, J., and Syid, U. (2004), The ACE Programmer's Guide, Addison-Wesley.
- [13] (2003), TAO Developer's Guide, Oci tao version 1.3a ed, Vol. 1 and 2, 12140 Woodcrest Executive Drive, Suite 250, St. Louis, MO, 63141: Object Computing Inc.
- [14] Bolton, F. (2002), Pure CORBA: A code intensive premium reference, SAMS.
- [15] Henning, M. and Vinoski, S. (1999), Advanced CORBA Programming with C++, Addison-Wesley.
- [16] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Sysetms for Robotics, An Applied Research Perspective, *International Journal of Advanced Robotic Systems*, Volume 3, 1(2005-204), 11–17.
- [17] Broten, G. and Collier, J. (2006), Continuous Motion, Outdoor, 2 1/2D Grid Map Generation using an Inexpensive Nodding 2-D Laser Rangefinder, In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, Number 2006-061, pp. pp 4240–4245, Orlando, FL.
- [18] Broten, G. and Collier, J. (2005), The Characterization of an Inexpensive Nodding Laser, (DRDC Suffield TR 2005-232) Defense R&D Canada – Suffield, Medicine Hat, Alberta.
- [19] Schmidt, D. and Kuhns, F. (2000), An Overview of the Real-time CORBA Specification, *IEEE Computer special issue on Object-Oriented Real-time Distrubuted Computing*.
- [20] Schmidt, D. and Huston, S. (2002), C++ Network Programming Volume 1, Addison-Wesley.
- [21] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Engineering for Experimental Robotics, Number DRDC Suffield SL 2005-227, Ch. UxV Software Systems, An Applied Research Perspective, Springer Tracts in Advanced Robotics.
- [22] Smith, Russell (2004), Open Dynamics Engine v0.5 Users Guide.
- [23] Koenig, S. and Likhachev, M. (2002), D* Lite, *Proceedings of the National Conference on Artificial Intelligence*, pp. 476–483.
- [24] Goldberg, S., Maimone, M., and Matthies, L. (2002), Stereo Vision and Rover Navigation Software for Planetary Exploration, *IEEE Aerospace Conference Proceedings*.
- [25] Singh, S., Schwehr, K., Simmons, R., Smith, T., Stenz, A., Verma, V., and Yahja, A. (2000), Recent Progress In Local and Global Traversability For Planetary Rovers, *International Conference on Robotics and Automation*.
- [26] Coulter, R. Craig (1992), Implementation of the Pure Pursuit Path Tracking Algorithm, (Tech Report CMU-RI-TR-92-01) Carnegie Mellon University.

- [27] Wheeler, D. (2004), SLOCCount: Source Lines of Code Count. Webpage. Version 2.26.

This page intentionally left blank.

Distribution list

DRDC Suffield TM 2006-188

Internal distribution

DRDC - Suffield

- 1 DG/DDG/Ch Sci/SMO
- 1 H/AISS
- 2 Lead Author
- 4 Other Authors (1 each)
- 1 Library - 1 hardcopy & 1 softcopy

Other DRDC

- 0 DRDKIM 1 softcopy

Total internal copies: 9

Total copies: 9

This page intentionally left blank.

DOCUMENT CONTROL DATA		
(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)		
1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Suffield PO Box 4000, Medicine Hat, AB, Canada T1A 8K6	2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable). UNCLASSIFIED	
3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title). Architecture for Autonomy		
4. AUTHORS (last name, first name, middle initial) Broten, G.S.; Collier, J.A.; Giesbrecht, J.L.; Monckton, S.P.; Mackay, D.J.		
5. DATE OF PUBLICATION (month and year of publication of document) December 2006	6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc). 64	6b. NO. OF REFS (total cited in document) 27
7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered). Technical Memorandum		
8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address). Defence R&D Canada – Suffield PO Box 4000, Medicine Hat, AB, Canada T1A 8K6		
9a. PROJECT NO. (the applicable research and development project number under which the document was written. Specify whether project).	9b. GRANT OR CONTRACT NO. (if appropriate, the applicable number under which the document was written).	
10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.) DRDC Suffield TM 2006-188	10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)	
11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification) (X) Unlimited distribution () Defence departments and defence contractors; further distribution only as approved () Defence departments and Canadian defence contractors; further distribution only as approved () Government departments and agencies; further distribution only as approved () Defence departments; further distribution only as approved () Other (please specify):		
12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).		

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

In 2002 Defence R&D Canada changed research direction from pure tele-operated land vehicles to general autonomy for land, air, and sea craft. The unique constraints of the military environment coupled with the complexity of autonomous systems drove DRDC to carefully plan a research and development infrastructure. This infrastructure, using a Component Based Software Engineering approach, would provide state of the art tools that didn't restrict the research scope; thus allowing DRDC to pursue its long-term research goals.

DRDC's long term objectives for its autonomy program address disparate unmanned ground vehicle (UGV), unattended ground sensor (UGS), air (UAV), and subsea and surface (UUV and USV) vehicles operating together with minimal human oversight (Collectively known as UxVs). The individual systems may range in complexity from simple reconnaissance mini-UAVs to sophisticated autonomous combat UGVs. These systems, when integrated into a common command and control structure that included manned elements, can provide long endurance, low risk battlefield services.

A key enabling technology for DRDC's autonomy research is a software architecture that meets both current and future requirements. DRDC adopted the Component Based Software Engineering philosophy to develop its software architecture known as the "Architecture for Autonomy". Although a well established practice in computing science, CBSE using frameworks has only recently entered common use in the field of UxV development. For industry and government, the complexity, cost, and time to re-implement stable systems often exceeds the perceived benefits of adopting a modern software infrastructure. Thus, most persevere with legacy software, adapting and modifying software when and wherever possible or necessary – adopting strategic software frameworks only when no justifiable legacy exists. Conversely, academic programs with short one or two year projects frequently exploit strategic software frameworks but with little enduring impact. Following the 2002 focus shift DRDC found itself in a unique position where researchers could freely review past experiences and latest advances in software technology, before selecting the best path forward. The open-source movement has a significant impact on DRDC's views with respect to software development. Academic frameworks, open to public scrutiny and modification, are now available for a variety of niche specific research areas and researchers leveraged this research to maximize the benefits to its autonomy research program.

This document describes the "Architecture for Autonomy", how it meets the program's current needs and details its usage on the Raptor UGV. It also presents an argument for why this architecture should also satisfy future requirements as well.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

architectures, Miro, ACE/TAO/CORBA, components, frameworks, middleware, unmanned ground vehicle, modularity, extensibility